DOCUMENT RESUME

ED 135 376 IR 004 484

AUTHOR	Burton, Richard R.
TITLE	Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	Naval Personnel Research and Development Lab., Washington, D.C.
REPORT NO	BBN-R-3453; ICAI-R-3
PUB DATE	Dec 76
CONTRACT	MDA903-76-C-0108
NOTE	145p.
EDRS PRICE	MF-$0.83 HC-$7.35 Plus Postage.
DESCRIPTORS	*Artificial Intelligence; Computer Assisted Instruction; *Computer Programs; Computer Science; *Man Machine Systems; *Programing Languages; *Semantics
IDENTIFIERS	SOPHIE

ABSTRACT

In an attempt to overcome the lack of natural means of communication between student and computer, this thesis addresses the problem of developing a system which can understand natural language within an educational problem-solving environment. The nature of the environment imposes efficiency, habitability, self-teachability, and awareness of ambiguity upon such a system. The major leverage points that allow these requirements to be met are limited domain, limited activities within that domain, and known conceptualization of the domain. Semantic grammar is introduced as a paradigm for organizing the knowledge required to understand and permit efficient phrasing. The need for succinct formalism for expressing semantic grammars led to the use of the Augmented Transition Networks (ATN). This led to the design and implementation of a general ATN compiling system which in turn translates an ATN into a program in a runnable computer language (LISP). The ATN compiler is also capable of producing programs which have been optimized to the features used by a particular ATN. The ability of ATN-based semantic grammars to perform satisfactorily in an educational environment is demonstrated in the natural language front-end for the SOPHIE system. (Author/WBC)

Documents acquired by ERIC include many informal unpublished materials not available from other sources. ERIC makes every effort to obtain the best copy available. Nevertheless, items of marginal reproducibility are often encountered and this affects the quality of the microfiche and hardcopy reproductions ERIC makes available via the ERIC Document Reproduction Service (EDRS). EDRS is not responsible for the quality of the original document. Reproductions supplied by EDRS are the best that can be made from the original.

ED135376

BBN Report No. 3453
ICAI Report No. 3

U.S. DEPARTMENT OF HEALTH.
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

SEMANTIC GRAMMAR: AN ENGINEERING TECHNIQUE FOR CONSTRUCTING NATURAL LANGUAGE UNDERSTANDING SYSTEMS*

By

Richard R. Burton

December 1976

Acknowledgements

I would like to thank Dr. John Seely Brown for his technical contributions throughout the course of this research and Dr. William Woods for fruitful discussions about ATN compilation.

The instructional and psychological aspects of this research were supported, in part, by the Advanced Research Projects Agency, Air Force Human Resources Laboratory, Army Research Institute, and Naval Personnel Research and Development Center under Contract No. MDA903-76-C-0108. The development of the ATN compiler was supported, in part, by Bolt, Beranek and Newman, Inc.

*A modified version of this report will appear as a technical report under the above mentioned contract.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Government.

IR 004484

ABSTRACT

One of the major stumbling blocks to more effective educational uses of computers is the lack of natural means of communication between the student and the computer. This thesis addresses the problem of developing a system which can understand natural language (English) within an educational problem-solving environment. The nature of the environment imposes the following requirements on a natural language understanding system: (1) efficiency, (2) habitability, (3) self-teachability, and (4) awareness of ambiguity. The major leverage points that allow these requirements to be met are: (1) limited domain, (2) limited activities within that domain, and (3) known conceptualizations of the domain. In other words, we know the problem area, the type of problem the student is trying to solve and the way he should be thinking about the problem in order to solve it.

The notion of semantic grammar is introduced as a paradigm for organizing the knowledge required to understand language which permits efficient parsing. In semantic grammar, non-terminal categories are formed on conceptual rather than syntactic bases. This allows semantic knowledge to be integrated into the parsing process whenever it is beneficial to do so. The semantic grammar also lends itself to a simple yet powerful method of handling pronominalizations, ellipses and other sentence fragments which arise naturally in a dialogue situation.

The need for a succinct formalism for expressing semantic grammars led to the use of the Augmented Transition Networks (ATN). This, in turn, led to the design and implementation of a general ATN compiling system which dramatically increases the speed of executing an ATN by translating it into a program in a runnable computer language (LISP). The ATN compiler is also capable of producing programs which have been optimized to the features used by a particular ATN. The ability of ATN-based semantic grammars to perform satisfactorily in an educational environment is demonstrated in the natural language front-end for the SOPHIE system.

Table of Contents

Chapter 1: Requirements for a Natural Language Interface 1
  Requirements 2

Chapter 2: Related Systems 7
  Keyword Schemes 7
  PARRY 8
  NLPQ 10
  CONSTRUCT 10
  RENDEZVOUS 11
  LUNAR 12
  Discussion 13

Chapter 3: Sample Dialogue 16

Chapter 4: Semantic Grammar 23
  Introduction 23
    Representation of Meaning 25
    Result of the Parsing 25
  Use of Semantic Information During Parsing 29
    Prediction 29
    Simple Deletion 30
    Ellipsis 31
  Using Context to Determine Referents 33
    Pronouns and Deletions 33
    Referents for Ellipses 35
    Limitations of the Context Mechanism 35
  Relationship to Other Semantic Systems 36
  Fuzziness 37
  Preprocessing 38
  Implementation 39

Chapter 5: Limitations of the LISP Implementation 41

Chapter 6: A Compiling System for Augmented Transition Networks . . . 46
  Introduction 46
  Augmented Transition Networks 47
    Developments to the ATN Formalism 50
  The General Notion of ATN Compilation 51
    Areas of Optimization 52
  A Grammar Compiling System 54
  A Version of the Grammar Compiling System 56
    Features of the ATN Machines 56
      Lexical Preprocessing 57
      Configurations 59
      Control Structure 60
    An Overview of an ATN Machine 61
    Anatomy of an ATN Machine 63

Code for the Arcs 64
WRD Arcs 65
CAT Arcs 66
PUSH Arcs 68
POP Arcs 69
Special Actions 70
Compiling the Compiled ATN 71
Results 71
Extensions to the Initial Version 72
The Well Formed Substring Table 73
Alternative Control Strategies 76

Chapter 7: Semantic ATN 77
Fuzziness 80
Comparison of Results 80

Chapter 8: Concluding Discussion 84
Impressions, Experiences and Observations 85
Research Areas in Semantic Grammar 86
Feedback - When the Grammar Fails 88
Future Research Areas in ATN Compilation 91
Conclusions 93

References 94

Appendix A: BNF Description of Part of the SOPHIE Semantic Grammar . . 98

Appendix B: A LISP Rule from the Semantic Grammar 100

Appendix C: Sample Parses and Parse Times for the LISP Implementation. 102

Appendix D: Examples of ATN Compilation. 104
Version I 107
Version II 111
Trace of Version I Parsing a Sentence. 117

Appendix E: Grammar Compiler Declarations. 121
Specification of Features 121
Declarations for Arc Tests and Actions 121

Appendix F: Debugging Features 124
Tracing 124
Breaks 124
How to Get into a Break 125
Grammar Break Commands 125

Appendix G: ATN Description of Part of the SOPHIE Semantic Grammar . . 127
Graphic Form of Semantic ATN 128
Input Form of Semantic ATN 131

Chapter 1

REQUIREMENTS FOR A NATURAL LANGUAGE INTERFACE

Since the inception of computing machines, a major problem has been facilitating man's communication with the machine. Much of the work which has been done in the areas of programming languages, natural language processing and automatic programming has been directed towards developing more natural man-machine interaction. This report discusses a paradigm for constructing efficient and friendly man-machine interface systems involving subsets of natural language for limited domains of discourse. As such this work falls somewhere between highly constrained formal language query system and unrestricted natural language understanding systems. The primary purpose of this research is not to advance our theoretical understanding of natural language but rather to put forth a set of techniques for embedding both semantic/conceptual and pragmatic information into a useful natural language interface module. Our intent is to produce a front end system which enables the user to concentrate on his problem or task rather than making him worry about how to communicate his ideas or questions to the machine.

Although there are many application areas for which our techniques apply, the principal motivation for this research arose from the pressing need for natural language interfaces to complex instructional systems underlying reactive educational environments. The term "educational environment" as used here refers to flexible problem solving, laboratory-like situations which have been implemented on a computer. The environment is reactive in the sense that the computer can (in addition to implementing the laboratory) monitor the student's activities and provide tutorial feedback at critical times during the solution of problems. Such systems have the characteristic that the computer naive users (students) are involved in a problem solving situation in which the computer is merely the medium. Most certainly these students (users) are not interested in state-of-art man-machine communication; they are interested in solving their problems and learning from their solution paths and errors.

The educational environment places constraints on a natural language understanding system which exceed the capabilities of all existing systems. These constraints include: (1) efficiency (2) habitability (3) self-teachability and (4) the ability to exist with ambiguity. In the remainder of this chapter we will explore why these are important and then provide an overview of the remainder of this report.

Requirements

A primary requirement for a natural language processor in a problem solving situation is speed. Imagine the following setting: the student is at his terminal actively working on a problem. He decides that he needs another piece of information to advance his solution so he formulates a query. From the time he finishes typing his question, he has nothing to do until the system gives him an answer so that he can continue working. The time the system spends parsing his query, the student is apt to spend forgetting pertinent information and losing interest. Psychological experiments have shown that response delays longer than two seconds have serious effects on the performance of complex tasks via terminals (Miller 68). In these two seconds, the system must understand the query; deduce, infer, lookup or calculate the answer; and generate a response.[1]

The second requirement for a natural language front-end is habitability. Any natural language system written in the forseeable future is not going to be able to understand all of natural language. What it must do is to characterize and understand a useable subset of the language. Watt defines a "habitable" sub-language as "one in which its users can express themselves without straying over the language boundaries into unallowed sentences" (1968 p. 338). Very intuitively, for a system to be habitable it must, among other things, allow the user to make local or minor modifications to an accepted sentence and get another accepted sentence. Exactly how much modification consitutes a minor change has never been specified. Some examples may provide more insight into the notion.[2]

---

[1] Another effect of poor response time which is critical to intelligent monitoring systems is that more of the student's searching for the answer is done internally (i.e. without using the system). This decreases the amount of information the system receives and increases the amount of induction that must be performed, making the problem of figuring out what the student is doing much harder (e.g. the student won't "show his work" when solving a problem; he will just give you the answer).

(1) Is anything wrong?
(2) Is there anything wrong?
(3) Is there something wrong?
(4) Is there anything wrong with section 3?
(5) Does it look to you like section 3 could have a problem?

If a problem solving system accepts sentence 1, it should also accept the modifications given in sentence 2 and 3. Sentence 4 presents a minor syntactic extension which may have major repercussions in the semantics but which should also be accepted. Sentence 5 is an example of a possible paraphrase of sentence 4 which is beyond the intended notion of habitability. Based on the acceptance of sentences 1-4, the user has no reason to expect that sentence 5 will be handled.

Any sub-language which does not maintain a high degree of habitability is apt to be worse than no natural language capability at all because the student will be forced to expend problem solving energies discovering how to formulate his question. That is, in addition to the problem he is seeking information about, the student is faced, sporadically, with the problem of getting the system to understand his query. This second problem can be disastrous both because it occurs seemingly randomly and because it is an ill-defined problem.[3]

---

[2] Similar examples are given in (Coles 1972) p. 262.

[3] In an informal experiment to test the habitability of a system the author asked a group of four students to write down as many ways as possible of asking a particular question. The original idea was to determine how many of the various paraphrasing would be accepted. The students each came up with one phrasing very quickly but had tremendous difficulty thinking of any others, even though three of the first phrasings were different! This experience demonstrates the lack of student's ability to do "linguistic" problem solving and points out the importance of accepting the student's first phrasing.

An equally important (and more challenging) aspect of the habitability problem is the problem of multi-sentence (or dialogue) phenomena. When students use a system which exhibits "intelligence" through its inference capabilities, they quickly start to assume that the system must also be intelligent in its conversational abilities as well. For example, they will frequently delete parts of their statements which they felt would be obvious given the context of the preceding statements. Often they are totally unaware of such deletions and show surprise and/or anger when the system fails to utilize contextual information as clearly as they (subconsciously) do. The use of context manifests itself in the use of such linguistic phenomena as pronominalizations, anaphoric deletions and ellipsis. The following sequence of questions exemplifies these problems:

(6) What is the population of Los Angeles?
(7) What is it for San Francisco?
(8) What about San Diego?

The third requirement for a natural language processor is that it be self-teaching (not learning, teaching). As the student uses the system, he should begin to feel the range and limitations of the sub-language. When the student does use a sentence that the system can't understand, he should receive insightful feedback which will enable him to figure out why. There are at least two kinds of feedback. The simplest (and most often seen) merely provides some indication of what parts of the sentence caused the problem (e.g. unknown word or phrase). A more useful kind of feedback goes on to provide a response based on those parts of the sentence that did make sense and then indicate (or give examples of) possibly related, acceptable sentences. It may even be advantageous to have the system recognize common sentences which would otherwise have been outside the sub-language and in response to them, explain why they are not in the sub-language.

The fourth requirement for a natural language system is that it be aware of ambiguity. Natural language gains a good deal of flexibility and power by not forcing every meaning into a different surface structure. This means that the program which interprets natural language sentences must be aware that its interpretation may not be the only one. For example, when asked:

(9) Was John believed to have been shot by Fred?

one of the most potentially disastrous responses is "Yes". The user may not be sure whether Fred did the shooting or the believing or both. More likely, the user, being unaware of any ambiguity, assumes one interpretation which may be different than the system's. If the system's interpretation is different, the user thinks he has the answer to his query when in fact he has the answer to a completely independent query. Either of the following is a much better response:

(10) Yes, it is believed that Fred shot John.
(11) Yes, Fred believes that John was shot.

Notice that the requirement is not that the system necessarily have tremendous disambiguation skills, but at least that it be aware that mis-interpretations are possible and inform the user of its interpretation. In those cases where the system makes a mistake the results may be annoying but should not be catastrophic.

This report presents the development of a technique, that we have named "semantic grammars", for building natural language processors which satisfy the above constraints. Chapter 2 discusses other systems which attack some of these problems. Chapter 3 presents a dialogue from the "intelligent" CAI system SOPHIE, which we used to refine and demonstrate this technique. This dialogue provides concrete examples of the kinds of linguistic capabilities that can be achieved using semantic grammars. Chapter 4 describes semantic grammars as they appear in SOPHIE, and points out how it allows semantic information to be used to handle dialogue constructs and to allow the directed ignoring of words in the input. Chapter 5 discusses the limitations which were encountered in the implementation of semantic grammars in the SOPHIE natural language processor and how these might be overcome by using a different formalism -- augmented transition networks. Chapter 6 describes a compiling system developed to improve the efficiency of the augmented transition network grammars by compiling them into runnable programs. Chapter 7 reports on the conversion of the SOPHIE semantic grammar to an ATN and the extensions to the ATN formalism which were necessary to maintain the solutions presented in chapter 4. Chapter 7 also includes comparison timing between the two versions of the natural language processor. Chapter 8 presents the conclusions of this research and suggests directions for further work.

Chapter 2

RELATED SYSTEMS

Much work has been done in the area of natural language understanding, and a number of different techniques have evolved from this research. For the purposes of this report, we shall limit the discussion of these systems to ones attempting to do question answering, and choose several systems that are most relevant to the problems with which we will be concerned. This chapter is not an attempt to review the entire field of natural language question answering systems (which has been done from many points of view (Simmons 1965,1969), (Wilks 1974), (Bates 1975), (Bruce 1975)), but, instead, to provide examples of practical systems throughout a range of complexity.

KEYWORD SCHEMES

Perhaps the oldest and simplest method of dealing with unrestricted natural language was through keyword parsing. The technique was introduced by Weizenbaum (1966a) and has been used and extended by others (see for example (Weizenbaum 1966b), (Brown et al 1973), (Shapiro et al 1975) (Colby et al 1974)). Using this parsing scheme, an input sentence is searched for "key" words. With each keyword is associated a collection of patterns which are tested against the complete input. If a pattern matches, an action associated with that pattern (typically a reassembly rule which constructs an output sentence by reassembling pieces of input) is executed. This action represents the "meaning" of the sentence to the system (i.e. the sentence's semantics).

Keyword analysis schemes have the advantage of being fast and of allowing the user great freedom of expression since any number of extraneous words can be included as long as the keywords appear. A particular parser can also be changed easily (by adding new rules) until such time as the rules begin interacting, at which point it is unclear which rule to use. When interactions do begin to occur, keywords can be assigned an "importance" number and the rule with the highest number can be used. However, conflicts may still arise when different keywords of the same importance appear in the same sentence.

Keyword techniques work well in situations where the actions that the system wishes to take in response to a sentence correspond in a simple way to the words (i.e. the concepts are not typically expressed as multiple word phrases, and words do not have multiple interpretations). However, they are weak in situations in which concepts are complex enough to require embedding or in which quantification[1] is required, since their semantic interpretation is essentially one level. In these cases, keyword patterns become more cumbersome and inefficient to use than more structural techniques. For example, consider the sentence:

(1) I think Q5 has an open emitter and a shorted base collector junction.

To recognize this sentence requires a very detailed keyword pattern which could be "keyed" equally well (equally poorly) off any of the words: think, Q5, open, emitter, shorted, base or collector. The main failing of the keyword technique is not being able to capture any of the structure of the language it is trying to characterize.

PARRY

PARRY is a ongoing project to develop a dialogue system that simulates paranoid behavior (Colby 1973), (Colby et al 1974). The system responds to any possible question and "understands" the questions well enough to exhibit paranoid behavior. To these ends, Colby has extended the keyword parsing techniques introduced by Weizenbaum by adding of a second level of matching. After a preprocessing phase collapses compound words, canonicalizes similar words, performs minor spelling correction and deletes

---

[1] Quantification refers to the problem of having a noun phrase which can range over a set of values, e.g. "some cars have engines", "all cars have engines". One of the problems with quantification is determining the scope of the quantification with respect to the rest of the sentence, especially when the rest of the sentence contains another quantifier.

unrecognized words, the input is segmented at certain keyword boundaries.[2] Each segment is then matched against a collection of segment patterns. The resulting list of recognized segments is then matched to a collection of complex patterns. Patterns have reassembly rules associated with them which construct the response.

Two important restrictions that should be placed on the application of keyword schemes to avoid mis-understandings (i.e. to avoid patterns applying when they shouldn't) have arisen from Colby's work. One is that at most one element should be ignored at each level of matching. Segment matches should account for all but one word. Complex patterns should account for all but one segment. The other restriction is that patterns should require that their elements occur in a particular order. The following example (from (Colby et al 1974)) demonstrates the usefulness of ignoring words (e.g. "well" in sentence 3), and the importance of word order (e.g. without word order restrictions, any pattern which matched 2 would also match 3).

(2) Are you well?
(3) Well, are you?

PARRY has demonstrated the capability of dealing with a relatively large number of concepts at a shallow level. The power in PARRY's approach lies in its ability to tolerate unknown words. As mentioned, this fuzziness is implemented by allowing the deletion of single elements from both levels of matching. Unfortunately the underlying semantics of PARRY's task, indeed the goals of the task itself, are vague, which makes attributes such as scope and habitability hard to evaluate. In addition, the two-level pattern matching technique lacks the precision required in a problem solving situation in which many regularities cannot be captured by one-level embedding.

---

[2] The fragmentation technique was developed by Wilks working in machine translation (1973a) (1973b). The list of segmentation words includes punctuation marks, subjunctives, conjunctions and prepositions.

Heidorn (Heidorn 1972,1974,1975) developed an automatic programming system called NLPQ which allows users to describe simulation problems in English. The system takes an English partial description of a problem and fits it into an internal description language, building pieces of the problem. From the partial internal description, questions are generated which request missing pieces of information. When the description is complete, the system can generate a GPSS program or an English description of the model it has built from the user's description. The user can also ask questions about the present model, and make changes and additions to it. The English processing is done using augmented phrase structure rules (discussed later). The phrase structure component is syntax-based (i.e. looks for things like noun phrases) with semantic restrictions being carried along in features which are tested in conditions on the phrase structure rules. The structure building augmentations create semantic/conceptual network structures, called segments, which represent the semantics of the phrase. Much of the system's success appears to be its close match between the structure of segments and the way English is used to describe modelling problems. No information on the use of NLPQ by naive users has been published, so it is difficult to evaluate the system's habitability.

CONSTRUCT

CONSTRUCT is a general system to do natural language processing developed at the Institute for Mathematical Studies in the Social Sciences (Smith et al 1974). Its major application is in a text-based, question answering system for elementary mathematics (Smith, N.W. 1974). The system answers questions such as:

(4) Are there any even prime numbers that are greater than 2?

(5) Is the sum of 5 and 2 less than the product of 5 and 2 but greater than the difference of 5 and 2?

The semantic basis of the system is a collection of procedures for generating and manipulating sets and numbers. The semantics of question 4 would be "are there any elements in the set created by intersecting the set of even numbers, the set of prime numbers and the set of numbers greater than 2?" As all of the sets in the example are infinite, the procedures know about dealing with intensional as well as extensional descriptions of sets.

The meaning of a sentence is determined by the following process. First a preprocess phase occurs during which (1) abbreviations are expanded, (2) synonyms are canonicalized, (3) compound word and common phrases are collapsed to a single word representation, (4) noise words are eliminated and (5) each word is replaced by its lexical category. The input is then parsed with a context free grammar with the semantic interpretation occurring in parallel via semantic construction functions associated with each grammar rule. Whereas this procedure is clearly inadequate if a traditional syntactic grammar is used, (e.g. no reasonable semantic function could be associated with the rule S := NP VP) the CONSTRUCT grammar is built around the semantic rules using categories which capture concepts in the application domain. For example, the grammar contains the grammatical category SUBST which corresponds to the semantic concept of a constructive set. This cuts across traditional category boundaries as seen in the sentences (from (Smith et al 1974)):

(6) Is 2 a factor of 4?
(7) How many factors of 12 are even?
(8) Give me the factors of 12 that are between 1 and 6.

The underlined portions would all be parsed into the SUBST category, although their traditional categories would be noun phrase, adjective, and prepositional phrase.

RENDEZVOUS

Codd is designing a natural language system, called RENDEZVOUS, to support the needs of casual users of data bases (Codd 1974). One problem that Codd has addressed, which has been neglected in previous systems, is what action to take if a user's query is beyond the restricted language understood by the system. A central notion to Codd's proposed solution to this problem is that of a "clarification dialogue" -- a system initiated dialogue that includes queries about an unacceptable utterance which attempts to arrive at the user's meaning. Codd points out that a clarification dialogue must be embarked upon very carefully. For example, if the system encounters the unknown word "concerning", one of the worst possible responses is "What do you mean by the word 'concerning'?" Almost any response to such a question would be beyond the capabilities of the system. Any clarification dialogue must be of "bounded scope" and guided by those parts of the query which the system can understand. RENDEZVOUS also employs re-statement of a user's query to confirm the intent of the query and to point out ambiguities. The range of language accepted by RENDEZVOUS, indeed even the method used to extend the range, is unclear. The aspect of RENDEZVOUS which is of interest here is the extent to which it has been designed as a "friendly" system.

LUNAR

The LUNAR system (Woods 1973a) (Woods et al 1972) is a natural language understanding implementation which combines a general semantic interpretation mechanism (Woods 1967,1968) with a large scale grammar of English (Woods 1970) (Woods et al 1972). LUNAR was designed to allow a lunar geologist to use English to query the chemical analysis data collected from the moon missions. Typical questions the system answers are:

(9) What is the average concentration of aluminium in high alkali rocks?

(10) Which samples have greater than 20% modal Plagioclase?

The processing of a query occurs in three major phases. During the first, the syntactic component derives the "deep structure" of the

---

[3] This is the linguistic deep structure hypothesized by Chomsky (Chomsky 1965) which has a central role in the theory of transformational grammar.

sentence.[3] The syntactic component uses a general transformational grammar of English syntax expressed as an augmented transition network (see Chapter 6). In the second phase a general, rule-driven semantic interpretation procedure produces the representation of the meaning of the sentence as a program in a formal retrieval language.[4] The semantic interpretation rules are tree-structured pattern matching rules which are used in groups to extract the meaning of different pieces of the syntax tree. The third phase is the execution of the formal expression to produce the answer to the request. The formal query language is a generalization of the predicate calculus which has been carefully designed to allow natural translation from English. The strength of the LUNAR system lies in its mechanisms to deal with quantification, conjunction, and relative clauses, and these are direct results of the carefully designed formal query language.

Discussion

The notion of an augmented phrase structure grammar provides a useful base for comparison between these systems.[5] An augmented phrase structure grammar contains two components. One is a set of context free phrase structure rules. The other is a corresponding set of functions (sometimes arbitrary, sometimes restricted) augmenting each of the rules which can be used to block the application of the context free rules and to maintain structures. While the paradigm of augmenting phrase structure grammars is followed by a large number of natural language systems, important differences exist with respect to what type of information is encoded in the grammar. For example, the LUNAR system uses a purely syntactic grammar[6]

---

[4] The notion that the meaning of a sentence is a program is generally called "procedural semantics". Procedural semantics is in general use for question answering applications. It does not, however, constitute a complete theory of meaning. In particular it does not account for such phenomena as declaratives, uses of temporal references, and belief structures.

---

[5] The idea of associating additional information with a phrase structure grammar has appeared in various forms since early compiling systems (Irons 1961).

and uses the augments to perform syntactic operations such as subject-verb agreement and to maintain the structure of the syntactic tree. NLPQ uses a syntactic grammar restricted by (usually semantic) features and uses the augments to perform parallel semantic interpretation. CONSTRUCT performs the semantic interpretation in parallel with a set of context free rules which are semantically oriented. PARRY's patterns, if viewed as limited phrase structure grammar rules, are directly linked to the semantics of the system. The decision about how much semantic information to encode in the grammar is a trade-off between efficiency and generality. Each of the systems presented here represents a defensible position along this spectrum.

When we began developing the SOPHIE system,[7] we explored the possibility of using intact the syntactic component of the LUNAR system. Since the LUNAR syntactic component was building a linguistically motivated description as opposed to the task oriented descriptions being built by the other systems, we felt its transferability to other domains would be high. We found the grammar to be very adequate parsing many of the most complicated sentences we felt SOPHIE would ever need to understand. Unfortunately, on simple sentences it provided more information about the sentence than we needed. For example, tense information was not needed or could be extracted from the relationships between concepts. The quantification and relative clause mechanisms were oriented towards Woods' formal query language which was not natural for our use. The use of conjunction in our domain is straightforward and relatively predictable, unlike its use in the LUNAR domain. All in all we had the feeling of using a microscope where we only needed a magnifying glass! The underlying

---

[6] The augmented transition network is an extension of a recursive transition network which has the power of a phrase structure grammar. For this reason we can classify it here as using an augmented phrase structure grammar. We will argue later that the transition network has conceptual advantages over phrase structure rules, but this does not affect this discussion which points out the difference in the kind of information captured in the grammar.

---

[7] A SOPHisticated Instructional Environment for teaching electronic troubleshooting. Chapter 3 provides examples of SOPHIE's language requirements.

semantic structure of our system just could not take advantage of such detail. Added detail is acceptable (it can always be ignored) except that the perception of such detail takes time which is a scarce commodity. At the time when we considered the LUNAR system, it was taking 2 or 3 seconds to syntactically parse a sentence and another 5 to semantically interpret it.[8] This experience led us to explore ways in which the semantics of the system could be used to speed the understanding process.

The technique we developed (described in Chapter 4) has much in common with both NLPQ and CONSTRUCT. However, significant differences arise from the emphasis we have placed on dealing with dialogues and on the construction of a friendly system. This has caused us to exploit two uses of semantics during parsing not found in these other systems. One is the insight provided into the nature of ellipsis and deletion in dialogues. The other is the basis provided for characterizing a habitable language. In Chapter 4, we shall discuss our concept of a semantic grammar and how it allows exploitation of these two advantages. Before we get into the details of how this is accomplished, we present in the next chapter an example of what has been accomplished.

---

[8] In Chapter 6 we will describe a technique which reduces the parse time by an order of magnitude making this approach more viable.

Chapter 3

SAMPLE DIALOGUE

Before delving into the structural aspects and technical details of the semantic grammar technique, we would first like to provide a concrete example of the dialogues it has supported. This chapter presents an annotated dialogue of a student using the "Intelligent" CAI system SOPHIE.[1] The dialogue is intended to demonstrate SOPHIE's linguistic capabilities and, while it touches upon the major features of SOPHIE, it is not meant to exhibit the logical or deductive capabilities the system.[2] In the dialogue, the student's typing is underlined. Even though the dialogue necessarily deals with electronic jargon, the linguistic issues it exemplifies occur in all domains. The annotations (lower case, indented) attempt to point out these problems and should be understandable to the non-electronics oriented reader.

WELCOME TO SOPHIE - A SIMULATED ELECTRONICS LABORATORY

> The circuit (Figure 3.1) is based on the Heathkit IP-28 power supply. The IP-28 is a reasonably sophisticated power supply with both current limiting and voltage limiting behavior. These two interrelated feedback loops make troubleshooting this circuit non-trivial.

>>INSERT A FAULT

> The student tells SOPHIE to give him a fault which he can troubleshoot. SOPHIE randomly selects a fault, inserts it into a

---

[1] SOPHIE was developed to explore the use of artificial intelligence techniques to provide tutorial feedback to students engaged in problem solving activities. The particular problem solving activity involved is that of troubleshooting a malfunctioning piece of electronic equipment. SOPHIE models the piece of equipment and answers the student's requests for measurements and other information to aid him in debugging. More important, throughout the problem solving session, SOPHIE has evaluate the logical consistency of a student's hypothesis or generate hypotheses which are consistent with the behavior the student has thus far observed.

---

[2] The reader is encouraged to see (Brown and Burton 1975) for further examples and descriptions of SOPHIE's tutorial and inferential capabilities.

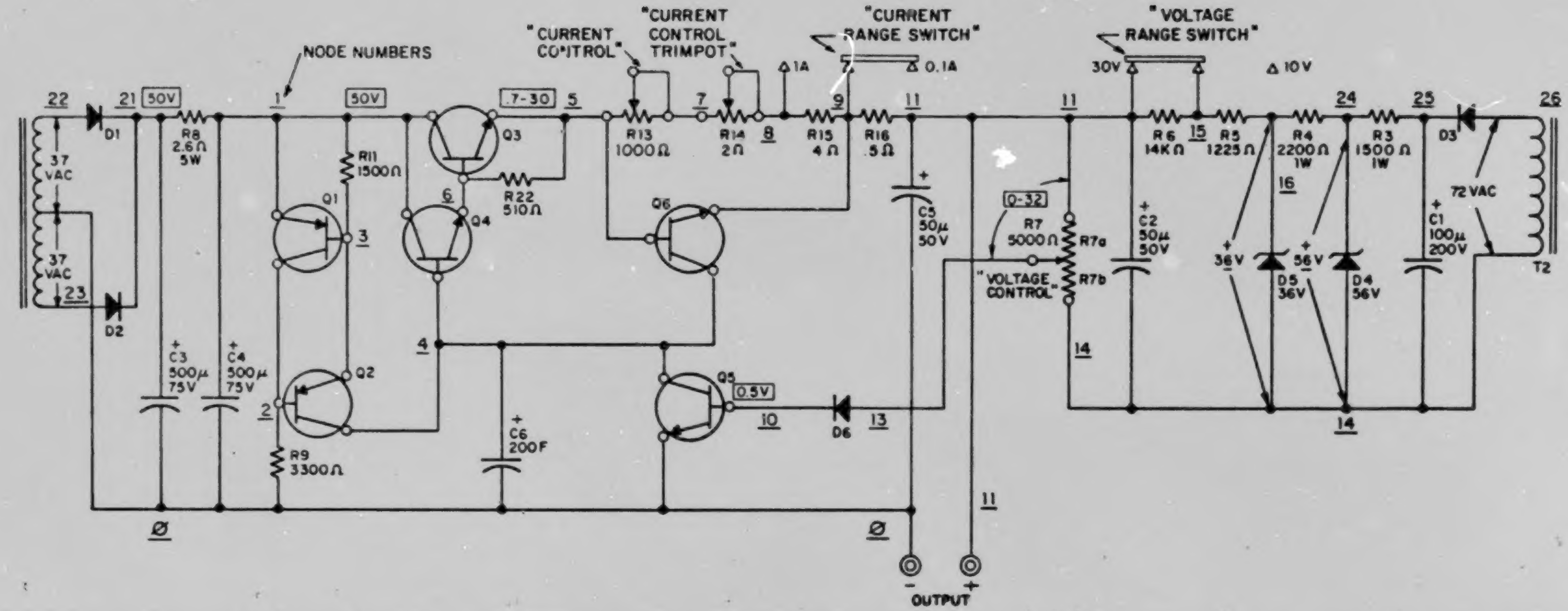

"CURRENT CONTROL"
"CURRENT CONTROL TRIMPOT"
"CURRENT RANGE SWITCH"
"VOLTAGE RANGE SWITCH"
NODE NUMBERS
1A
0.1A
30V
10V
22
21
50V
1
50V
.7-30
5
7
8
9
11
11
15
24
25
26
37 VAC
37 VAC
23
D1
D2
R8 2.6Ω 5W
R11 1500Ω
Q1
3
Q2
2
R9 3300Ω
C3 500µ 75V
C4 500µ 75V
Q3
Q4
6
R22 510Ω
4
C6 200F
R13 1000Ω
R14 2Ω
R15 4Ω
R16 .5Ω
Q6
Q5
0.5V
10
D6
13
C5 50µ 50V
0-32
R7 5000Ω
R7a
R7b
"VOLTAGE CONTROL"
14
R6 14KΩ
R5 1225Ω
R4 2200Ω 1W
R3 1500Ω 1W
16
C2 50µ 50V
36V
D5 36V
56V
D4 56V
14
C1 100µ 200V
72 VAC
D3
T2
Ø
Ø
11
OUTPUT
-
+

Figure 3.1
Circuit Schematic of Heathkit IP-28 Power Supply

model of the instrument and tells the student how the front panel controls are presently set.

```
THE INSTRUMENT HAS A FAULT AND ITS PRESENT CONTROL SETTINGS ARE:
CC        1.0     CURRENT CONTROL-FROM 0.0 (MIN CURRENT) TO 1.0
CR        HIGH    CURRENT RANGE SWITCH, HIGH=1 AMP, LOW=.1 AMP
LOAD      1000    LOAD RESISTANCE IN OHMS
VC        1.0     VOLTAGE CONTROL-FROM 0.0 (MIN VOLTAGE) TO 1.0
VR        HIGH    VOLTAGE RANGE SWITCH, HIGH=30, LOW=10 VOLTS
```

```
>>WHAT IS THE OUTPT?
OUTPT => OUTPUT
THE OUTPUT VOLTAGE IS 1.2 VOLTS.
```

SOPHIE corrects the student´s typing mistake. Notice that "the output" was disambiguated as the output voltage.

```
IF YOU ARE NOT SURE WHETHER A VALUE IS CORRECT, YOU CAN ASK "IS THAT
VALUE CORRECT?"
```

The system points out a feature that the student may not otherwise be aware of. "Correct" here means what it is in a working instrument.

```
>>IS THAT RIGHT
IN A WORKING CIRCUIT THE OUTPUT VOLTAGE IS 30.1 VOLTS.
THAT VALUE IS INCORRECT.
```

"That" here refers to the value of the previous measurement. SOPHIE, in addition to the faulted instrument, has a working circuit which can be used to take reference measurements.

```
>>WHAT IS THE VOLTAGE BETWEEN THE CURRENT LIMITING TRANSISTOR
...AND THE CONSTANT CURRENT SOURCE?
THE VOLTAGE AT N4 IS 2.48 VOLTS.
```

In addition to specific part names, SOPHIE recognizes references to parts by their section. Notice that even though the sentence says "voltage between x and y", and it really means "voltage at z with respect to ground". The output routines rephrase the input in a simpler form (i.e. between x and y reduces to N4).

```
>>WHAT IS IT BETWEEN THERE AND THE BASE OF Q6
THE VOLTAGE BETWEEN N4 AND N5 IS 1.28 VOLTS.
```

Here "it" refers to "the voltage" and "there" refers to N4. Also notice that "between" has to be treated differently, i.e. two nodes are explicit now.

```
>>IS THERE ANY RIPPLE IN THE VOLTAGE AT THE BASE OF Q4?
I´M SORRY, I DO NOT HANDLE AC CHARACTERISTICS.
```

SOPHIE recognizes that the student is moving into an area which it is unable to handle. This is currently done by keyword recognition (e.g. ripple) but points out a technique which should be incorporated in other aspects of the system as well.

```
>>WHAT IS THE BASE CURRENT OF Q3
THE BASE CURRENT OF Q3 IS 12 MICROAMPS.
```

SOPHIE can answer questions about current, resistance, and power dissipation as well as voltage. The output and input routines know about unit prefixes.

```
>>THRU THE EMITTER?
THE EMITTER CURRENT OF Q3 IS -.26 MILLIAMPS.
```

This is a type of sentence fragment which occurs frequently in dialogues which we will refer to as an ellipsis. Rather than specifying a complete thought, an ellipsis only gives that piece of the thought which contrasts an earlier thought. Notice that the referent cannot be figured out on syntactic grounds (as is possible with the ellipsis "the emitter current"; yet this is the sort of statement a student might make if the last answer had caused a slight shifting of his thinking.

```
>>COULD THE PROBLEM BE THAT Q5 IS BAD?
PLEASE BE MORE SPECIFIC ABOUT HOW Q5 IS FAULTED.
```

The student develops an idea about what is wrong and asks to have it evaluated.

```
>>COULD IT BE SHORTED?
THE SHORTED MODES OF A TRANSISTOR CURRENTLY IMPLEMENTED ARE: ALL
JUNCTIONS SHORTED OR ONE JUNCTION SHORTED WITH THE OPPOSITE
TERMINAL MODELLED OPEN.
```

A hypothesis must be completely (or non-ambiguously) specified before evaluation can take place. Even though the last statement generated an error condition, it is still used to determine possible pronoun referents.

```
SHOULD Q5 HAVE ALL JUNCTIONS SHORTED?
> NO
WHICH TERMINAL SHOULD BE MODELLED OPEN?
> THE BASE
OK, I'LL LET Q5 HAVE ITS EMITTER COLLECTOR JUNCTION SHORTED AND ITS BASE
 OPEN
```

At this point, SOPHIE looks back over the student's measurements to determine if he should have "known better". See (Brown and Burton 1975) or (Brown et al 1975) for further description.

```
HMMM, LET ME COMPUTE ON THAT FOR A MINUTE.
```

When the system knows there may be a long wait, a "delaying" message is printed to let the student know something is happening.

```
WHEN THE SETTINGS WERE:
        CC=1.0, CR=HIGH, LOAD=1000, VC=1.0, VR=HIGH,
THAT FAULT CONFLICTS WITH THE FOLLOWING MEASUREMENT.
YOU OBSERVED THAT THE VOLTAGE BETWEEN N4 AND N5 WAS 1.28 VOLTS.
IF Q5 HAD ITS EC SHORTED AND ITS BASE OPEN IT WOULD BE 1.1 MILLIVOLTS.
IN A WORKING CIRCUIT IT IS 1.43 VOLTS.
```

```
BUT THAT FAULT DOES EXPLAIN THE VALUES YOU OBSERVED FOR
THE OUTPUT VOLTAGE, THE VOLTAGE AT N4, THE VOLTAGE BETWEEN N4 AND N10,
THE EMITTER CURRENT OF Q3 AND THE BASE CURRENT OF Q3.
```

```
>>WHAT COULD BE WRONG
```

The student becomes stuck and asks for assistance. SOPHIE proposes some possible hypotheses for the student to explore.

```
THERE ARE LOTS OF FAULTS WHICH WOULD EXPLAIN THE MEASUREMENTS YOU
HAVE TAKEN. HERE ARE SOME POSSIBILITIES:
THE RESISTANCE OF R8 CHANGES
THE LEAKAGE RESISTANCE OF C4 CHANGES
THE BETA OF Q3 CHANGES
R15 IS OPEN
Q6 IS FAULTED

>>IF R8 WERE 30K, WHAT WOULD THE OUTPUT VOLTAGE BE?
IF THE VALUE OF R8 WERE 30000
UNDER THE PRESENT SETTINGS, THE OUTPUT VOLTAGE WOULD BE .904 VOLTS.
IN A WORKING CIRCUIT IT IS 30.1 VOLTS.
```

The student doesn't believe one of the possiblities so he checks it using a hypothetical statement. Even though the voltage in this case and that in the case of a faulted circuit are not the same, they qualitatively agree.

```
>>REPLACE Q6
```

The student guesses maybe Q6 is bad and tries to replace it. When the student replaces a part SOPHIE queries him to insist that he know how it is faulted.

```
I AM GOING TO ASK YOU SOME QUESTIONS ABOUT HOW Q6 IS FAULTED.
ARE ANY JUNCTIONS SHORTED?
> YES
WHICH JUNCTIONS ARE SHORTED?
> ALL
```

If the response to a query is not understood, further specification of the expected response is given.

```
PLEASE SPECIFY THE JUNCTIONS AS EC EB AND CB (SEPARATED BY SPACES).
WHICH JUNCTIONS ARE SHORTED?
> EC EB AND CB
Q6 IS NOT FAULTED IN THAT MANNER.
```

The part would be replaced if any of the junctions had been shorted. SOPHIE would normally have critiqued the student's hypothesized fault. This was not included in the dialogue to save space.

```
Q6 NOT REPLACED.

>>VOLTAGE DROP ACROSS R5
THE VOLTAGE ACROSS R5 IS -.028 MILLIVOLTS.
```

Some noun phrase utterances are recognized as questions.

```
>>WHAT ABOUT R3?
THE VOLTAGE ACROSS R3 IS 71.6 VOLTS.
```

Another ellipsis. The context of the previous question is used to determine "what" about R3.

```
>>REPLACE R3
IS R3 OPEN OR SHORTED OR DOES IT HAVE THE WRONG VALUE?
> IT IS OPEN
R3 REPLACED
```

```
>>IS THERE ANYTHING ELSE WRONG
NO
```

After fixing the fault, the student uses SOPHIE to explore other aspects of the circuit's behavior.

```
>>IF C2 WERE LEAKY, WHAT WOULD THE VOLTAGE ACROSS IT BE?
```

The referent to "it" in this case occurs earlier in the same sentence.

```
IS IT OKAY IF I USE 10000 FOR THE LEAKAGE RESISTANCE OF C2?
> YES
IF THE LEAKAGE RESISTANCE OF C2 WERE 10000
UNDER THE PRESENT SETTINGS, THE VOLTAGE ACROSS C2 WOULD BE 26.3 VOLTS.
IN A WORKING CIRCUIT IT IS 28.9 VOLTS.
```

```
>>WHAT HAPPENS IF C1 SHORTS?
UNDER THE SETTINGS:
        CC=1.0, CR=HIGH, LOAD=1000, VC=1.0, VR=HIGH,
IF C1 WERE SHORTED, D3 WOULD BECOME OVERLOADED.
```

In the case where nothing more interesting happens, "what happens" questions default to the output voltage.

```
>>IF Q6 HAS AN OPEN EMITTER AND A SHORTED BAE COLLECTOR JUNCTION
...WHAT HAPPENS TO THE VOLTAGE BETWEEN ITS BASE AND THE JUNCTION OF
...THE VOLTAGE LIMITING SECTION AND THE VOLTGEA REFERENCE SOURCE?
BAE => BASE
VOLTGEA => VOLTAGE
```

Several examples of the use of conjunction. Imagine the student's frustration if he is forced to type this query over because of spelling mistakes.

```
IF Q6 HAD ITS BASE COLLECTOR JUNCTION SHORTED AND ITS EMITTER OPEN,
UNDER THE PRESENT SETTINGS:
        CC=1.0, CR=HIGH, LOAD=1000, VC=1.0, VR=HIGH
THE VOLTAGE BETWEEN N5 AND N13 WOULD BE THE SAME, I.E. 28.9 VOLTS.
```

A hypothetical question is first tried under the present settings. If no difference is noted, other settings are tried.

```
LET ME TRY ANOTHER CASE WHICH USUALLY POINTS OUT PROBLEMS IN THAT
SECTION OF THE CIRCUIT.
UNDER THE SETTINGS:
        CC=.9, CR=LOW, LOAD=10, VC=1.0, VR=LOW,
THE VOLTAGE BETWEEN N5 AND N13 WOULD BE 8.94 VOLTS.
IN A WORKING CIRCUIT IT IS 9.57 VOLTS.

>>WHAT IS THE OUTPUT VOLTAGE WHEN THE VOLTAGE CONTROL IS SET TO .5?
WITH THE VOLTAGE CONTROL NOW SET AT .5
THE OUTPUT VOLTAGE IS 15.6 VOLTS.
```

A hypothetical question can involve the changing of a control. If this happens the control is left at the new setting and the student is told so.

```
>>IF IT IS SET TO .7?
WITH THE VOLTAGE CONTROL NOW SET AT .7
THE OUTPUT VOLTAGE IS 21.4 VOLTS.
```

Context determines which measurement should be taken and which control should be set.

```
>>.9?
WITH THE VOLTAGE CONTROL NOW SET AT .9
THE OUTPUT VOLTAGE IS 27.2 VOLTS.

>>WHAT IS THE CURRENT THRU THE CC WHEN THE VC IS SET TO 1.0?
WITH THE VOLTAGE CONTROL NOW SET AT 1.0
THE CURRENT THRU THE CURRENT CONTROL IS 30 MILLIAMPS.
```

CC and VC are abbreviations. Notice that controls are also "parts" which can be used in measurements.

```
>>WHAT IS IT WITH IT SET TO .8?
WITH THE VOLTAGE CONTROL NOW SET AT .8
THE CURRENT THRU THE CURRENT CONTROL IS 24 MILLIAMPS.
```

Here the system must correctly instantiate two occurrences of "it". The referent to the second "it" could be either CC or VC since these are both controls.

```
>>GOODBYE
GOODBYE
```

Chapter 4

SEMANTIC GRAMMAR

INTRODUCTION

In Chapter 1 we described the requirements for a natural language processor in a learning environment. Briefly, they are efficiency and friendliness over the class of sentences which arise in a dialogue situation. The major leverage points we have that allow us to satisfy these requirements are (1) limited domain, (2) limited activities within that domain, and (3) known conceptualizations of the domain. In other words, we know the problem area, the type of problem the student is trying to solve, and the way he should be thinking about the problem in order to solve it. What we are then faced with is taking advantage of these constraints in order to provide an effective communication channel.

Notice that all of these constraints relate to concepts underlying the student´s activities. In SOPHIE, the concepts include voltage, current, parts, transistors, terminals, faults, particular parts (e.g. R9, Q5, etc.), hypotheses, controls, settings of controls, and so on. The (dependency) relationships between concepts include things like voltage can be measured at terminals, parts can be faulted, controls can be set, etc. The student, in formulating a query or statement, is requesting information or stating a belief about one of these relationships, e.g. "What is the voltage at the collector of Q5" or "I think R9 is open". It occurred to us that the best way to characterize the statements used for this task was in terms of the concepts themselves as opposed to the traditional syntactic structures. The language can be described by a set of grammar rules which characterize, for each concept or relationship, all of the ways of expressing it in terms of other constituent concepts. For example, the concept of a measurement requires a quantity to be measured and something to measure it with respect to. A measurement is typically expressed by giving the quantity followed by a preposition followed by the thing that specifies where to measure, e.g. "voltage across C2", "current thru D1", "power dissipation of R9", etc. These phrasings are captured in the grammar rule:[1]

<MEASUREMENT> := <MEASUREABLE/QUANTITY> <PREP> <PART>

The concept of a measurement can, in turn, be used as part of other concepts, e.g. to request a measurement "What is the voltage across C2?"; or to check a measurement "Is the current thru D1 correct?". We will call this type of grammar a "semantic grammar" because the relationships it tries to characterize are semantic/conceptual as well as syntactic.

Semantic grammars have two advantages over traditional syntactic grammars. They allow semantic constraints to be used to make predictions during the parsing process, and they provide a useful characterization of those sentences which the system should try to handle. The predictive aspect is important for four reasons. (1) It reduces the number of alternatives which must be checked at a given time; (2) it reduces the amount of syntactic (grammatical) ambiguity; (3) it allows recognition of ellipsed or deleted phrases; and (4) it permits the parser to skip words at controlled places in the input (i.e. it enables a reasonable specification of control). These points will be discussed in detail in a later section.

The characterization aspect is important for two reasons. (1) It provides a handle on the problem of constructing a habitable sub-language. The system knows how to deal with a particular set of tasks over a particular set of objects. The sub-language can be partitioned by tasks to accept all straightforward ways of expressing those tasks, but does not need to worry about others. (2) It allows a reduction in the number of sentences which must be accepted by the language while still maintaining habitability. There may be syntactic constructs which are used frequently with one concept (task) but seldom with another. For example, relative clauses may be useful in explaining the reasons for performing an experimental test but are an awkward (though possible) way of requesting a measurement. By separating the processing along semantic grounds, one may gain efficiency by not having to accept the awkward phrasing.

---

[1] This is not actually a rule from the grammar but is merely intended to be suggestive.

Representation of Meaning

Since natural language communication is the transmission of concepts via phrases, the "meaning" of a phrase is its correspondent in the conceptual space. The entities in SOPHIE's conceptual space are objects, relationships between objects, and procedures for dealing with objects. The meaning of a phrase can be a simple data object (e.g. "current limiting transistor") or a complex data object (e.g. "C5 open", "Voltage at node 1"). The meaning of a question is a call to a procedural specialist which knows how to determine the answer. The meaning of a command is a call to a procedure which performs the specified action.[2] For example, the procedural specialist DOFAULT knows how to fault the circuit and is used to represent the meaning of commands to fault the circuit (e.g. "Open R9", "Suppose C2 shorts and R9 opens"). The concept (argument) that DOFAULT needs in order to perform its task is an instance of the concept of faults which specifies the particular changes to be made, e.g. "R9 being open". These same concepts of particular faults also serve as arguments to two other specialists: HYPTEST which determines the consistency of a fault with respect to the present context, e.g. "Could R9 be open"; and SEEFAULT which checks the actual status of the circuit, e.g. "Is R9 open?".

Result of the Parsing

Basing the grammar on conceptual entities allows the semantic interpretation (determination of the concept underlying a phrase) to proceed in parallel with the parsing. Since each of the non-terminal categories in the grammar is based on a semantic unit, each grammar rule can specify the semantic description of a phrase that it recognizes in much the same way that a syntactic grammar specifies a syntactic description. The construction portion of the rules is procedural so that each rule has the freedom to decide how the semantic descriptions, returned by the constituent items of that rule, are to be put together to form the correct "meaning".

---

[2] Declarative statements are treated as requests because the pragmatics of the situation imply that the student is asking for verification of his statement. For example, "I think C2 is shorted" is taken to be a request to have the hypothesis "C2 is shorted" critiqued.

For example, the meaning of the phrase "Q5" is the data base object Q5. The meaning of the phrase "the collector of Q5" is (COLLECTOR Q5) where COLLECTOR is a function which returns the data base item which is the collector of the given transistor. For a more complicated example, consider the non-terminal <MEASUREMENT> shown in Figure 4.1. The goal for this non-terminal is to capture all of the ways that a student can specify a measurement (voltage across D3, output current, etc.). To specify a measurement, there must be a quantity to be measured <MEAS/QUANT> (voltage, current, resistance, power dissipation), and something to measure with respect to (e.g. a part, <PART/SPEC>; a transistor junction, <JUNCTION>;

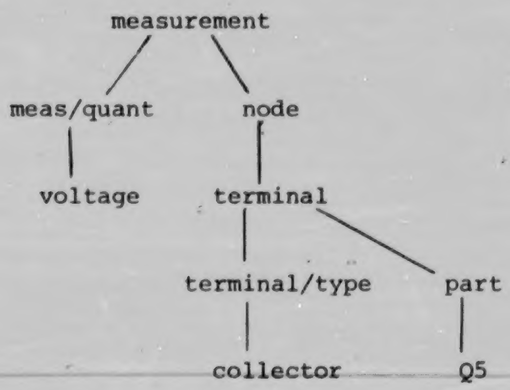
Figure 4.1[3]
A Semantic Grammar Rule

```
<MEASUREMENT> ::= output <MEAS/QUANT> [of <TRANSFORMER>] !
                 <TRANSFORMER> <MEAS/QUANT> !
                 <MEAS/QUANT> between <NODE> and <NODE> !
                 <MEAS/QUANT> <PREP> <PART> !
                 <MEAS/QUANT> between output terminals !
                 <MEAS/QUANT> <PREP> <JUNCTION> !
                 <MEAS/QUANT> <PREP> <NODE> !
                 <JUNCTION/TYPE> <MEAS/QUANT>
                      of <TRANSISTOR/SPEC> !
                 <TRANSISTOR/TERM/TYPE> <MEAS/QUANT>
                      of <TRANSISTOR>
```

---

[3] The rule is expressed in a BNF-like notation which is an abstraction of the actual rule (see next section). Non-terminals are in capital letters and enclosed in angle brackets. Terminal are in lower case. Brackets enclose optional elements. Alternative right hand sides are separated by a "!".

or possibly a point in the circuit, <NODE>). The rule for <MEASUREMENT> expresses all of the ways that the student can give a measurable quantity and also supply its required arguments. The structure which results from <MEASUREMENT> is a function call to the function MEASURE which supplies the quantity being measured and other arguments specifying where to measure it. Thus the meaning of the phrase "the voltage at the collector of Q5" is (MEASURE VOLTAGE (COLLECTOR Q5)) which was generated from the control structure:



A careful examination of Figure 4.1 reveals that <MEASUREMENT> also accepts "meaningless" phrases such as "the power dissipation of Node 4." In addition, it accepts some meaningful phrases such as "the resistance between Node 3 and Node 14" which SOPHIE does not calculate. This results from generalizing together concepts which are not treated identically in the surface structure. In this case voltage, current, resistance and power dissipation were generalized to the concept of a measurable quantity. Allowing the grammar to accept more statements and having the argument checking done by the procedural specialists has the advantage of allowing the semantic routines to provide the feedback as to why a sentence cannot be interpreted or "understood". It also keeps the grammar from being cluttered with special rules for blocking meaningless phrases. Carried to the limit, the generalization strategy would return the grammar to being "syntactic" again (e.g. all data objects are "noun phrases"). The trick is to leave semantics in the grammar when it is beneficial to do so (i.e. to stop extraneous parsings early or tighten the range of a referent for an ellipsis or deletion). This is obviously a task-specific trade-off.[4]

The relationship between a phrase and its meaning is usually straightforward. However, it is not limited to simple embedding. Consider the phrases "the base emitter of Q5 shorted" and "the base of Q5 shorted to the emitter". The thing which is "shorted" in both of these phrases is the "base emitter junction of Q5." The rule which recognizes both of these phrases, <PART/FAULT/SPEC>, can handle the first phrase by invoking its constituent concepts of <JUNCTION> (base emitter of Q5) and <FAULT/TYPE> (shorted) and combining their results. In the second phrase, however, it must construct the proper junction from the separate occurrences of the two terminals involved. Figure 4.2 gives the rules used to recognize these two situations. The situations are distinguished by the occurrence of the optional constituent in the second phrase. (As will be discussed later, the rules are procedurally encoded which provides a natural way of building separate semantic forms for the two cases.) Notice that the parser does some paraphrasing, as the "meaning" of the two phrases is the same.

The discussion has been presented as if the concepts were defined a priori by the capabilities of the system. Actually, for the system to remain at all habitable, the concepts are discovered in the interplay between the statements that are made in the domain and the capabilities of the system. When a particular English construct is difficult to handle, it is probably an indication that the concept it is trying to express has not

Figure 4.2
Grammar Rules

```
<PART/FAULT/SPEC> := <FAULTABLE/THING> is <FAULT/TYPE>
                         [to <TRANSISTOR/TERMINAL/TYPE>]

<FAULTABLE/THING> := <JUNCTION> ! <TERMINAL> ! <PART>

<FAULT/TYPE> := open ! shorted

<TRANSISTOR/TERMINAL/TYPE> := base ! emitter ! collector
```

---

[4] Bobrow and Brown (1975) describe an interesting paradigm from which to consider this trade-off.

been recognized properly by the system. In our example "the base of Q5 is shorted to the emitter", the relationship between the phrase and its meaning is awkward because the present concept of shorting requires a part or a junction. The example is getting at a concept of shorting in which any two terminals can be shorted together (e.g. "the positive terminal of R9 is shorted to the anode of D6"). This is a viable conceptual view of "shorting", but its implementation requires allowing arbitrary changes in the topology of the circuit which is beyond the efficiency limitations of SOPHIE's simulator. Thus, the system we were working with led us to define the concept in too limited a way.

USE OF SEMANTIC INFORMATION DURING PARSING

Prediction

Having described the notion of a semantic grammar, we now describe the ways it allows semantic information to be used in the understanding process. One use of semantic grammars is to predict the possible alternatives that must be checked at a given point. Consider for example the phrase "the voltage at xxx" (e.g. "the voltage at the junction of the current limiting section and the voltage reference source"). After the word "at" is reached in the top-down, left-to-right parse, the grammar rule corresponding to the concept "measurement" can predict very specifically the conceptual nature of "xxx", i.e. it must be a phrase which directly or indirectly (e.g. the junction of the current limiting section and the voltage reference source) specifies a location (e.g. node, terminal, etc.) in the circuit.

Semantic grammars also have the effect of reducing the amount of grammatical ambiguity. In the phrase "the voltage at xxx", the prepositional phrase "at xxx" will be associated with the noun "voltage" without considering the potentially ambiguous parse which associates it someplace higher in the tree.

Predictive information is also used to aid in the determination of referents for pronouns. If the above phrase were "the voltage at it", the grammar would be able to restrict the class of possible referents to locations. By taking advantage of the available sentence context to predict the semantic class of possible referents, the referent determination process is greatly simplified. For example:

(1a) Set the voltage control to .8?
(1b) What is the current thru R9?
(1c) What is it with it set to .9?

In (1c), the grammar is able to recognize that the first "it" refers to a measurement (that the student would like re-taken under slightly different conditions). The grammar can also decide that the second "it" refers to either a potentiometer or to the load resistance (i.e. one of those things which can be set). The referent for the first "it" is the measurement taken in (1b), "the current thru R9". The referent for the second "it" is "the voltage control" which is an instance of a potentiometer. The context mechanism which selects the referents will be discussed later.

Simple Deletion

The semantic grammar is also used to recognize simple deletions. The grammar rule for each conceptual entity knows the nature of that entity's constituent concepts. When a rule cannot find a constituent concept, it can either

i) fail (if the missing concept is considered to be obligatory in the surface structure representation) or

ii) hypothesize that a deletion has occurred and continue.

For example, the concept of a TERMINAL has (as one of its realizations) the constituent concepts of a TERMINAL-TYPE and a PART. When its grammar rule only finds the phrase "the collector", it uses this information to posit that a part has been deleted (i.e. TERMINAL-TYPE gets instantiated to "the collector" but nothing gets instantiated to PART). The natural language processor then uses the dependencies between the constituent concepts to determine that the deleted PART must be a TRANSISTOR. The "meaning" of this phrase is then "the collector of some transistor". Which transistor is determined when the meaning is evaluated in the present dialogue context. In particular, the semantic form returned is the function PREF and the classes of possible referents; in our example the form would be (COLLECTOR (PREF ´(TRANSISTOR))).[5] The operation of PREF will be discussed later.

Ellipsis

Another use of the semantic grammar allows the processor to recognize elliptic utterances. These are utterances which do not express complete thoughts (i.e. a completely specified question or command) but only give differences between the intended thought and an earlier one.[6] For example, 2b, 2c and 2d are elliptic utterances.

(2a) What is the voltage at Node 5?
(2b) At Node 1?
(2c) and Node 2?
(2d) What about between nodes 7 and 8?

Ellipses can begin with introductory phrases such as "and" in 2c or "what about" in 2d; however this is not required as can be seen in 2b. Part of

Figure 4.3

Ellipsis Rule

```
<ELLIPSIS> := [<ELLIPSIS/INTRODUCER>] <REQUEST/PIECE> !
              [<ELLIPSIS/INTRODUCER>] if <PART/FAULT/SPEC>

<REQUEST/PIECE> := [<PREP>] <NODE> !
                   [<PREP>] <PART> !
                   between <NODE> and <NODE> !
                   [<PREP>] <JUNCTION> !
                   etc.
```

---

[5] The language LISP will be used in examples throughout this thesis. In LISP, a function call is expressed in Cambridge-Polish notation: as a parenthesized list of the function name followed by its arguments.

[6] The standard use of the word "ellipsis" refers to any deletion. Rather than invent a new word, we shall use the restricted meaning here.

the ellipsis rule is given in Figure 4.3. The grammar rule identifies which concept or class of concepts are possible from the context available in the elliptic utterance.

While the parser is usually able to determine the intended concepts from the context available in an elliptic utterance, this is not always the case. Consider the following two sequences of statements.

(3a) What is the voltage at Node 5?
(3b) 10?

(4a) What is the output voltage if the load is 100?
(4b) 10?

In (3b), "10" refers to node 10, while in (4b) it refers to a load of 10. The problem this presents to the parser is that the concepts underlying these two elliptic utterances have nothing in common except their surface realizations. The parser, which operates from conceptual entities, does not have a concept which includes both of these interpretations. One solution would be to have the parser find all parses (concepts) and then choose between them on the basis of context. Unfortunately, this has the unacceptable property that time is spent looking for more than one parse for the large percentage of sentences in which it is unnecessary. A better solution would be to allow structure among the concepts so that the parser would recognize "10" as a member of the concept number, and then the routines which find the referent would know that numbers can be either node numbers or values. This type of recognition could profitably be performed by a bottom-up approach to parsing. However, its advantages over the present scheme are not enough to justify the expense incurred by a bottom-up parse finding all possible well-formed constituents. At present, the parser assumes one interpretation and a message is printed to the student indicating the assumed interpretation. If it is wrong, the student must supply more context in his request. In fact, "10?" is taken as a load specification and if the student meant the node he would have to use "at 10", "N10" or "Node 10". Later we will discuss the mechanism that determines to which complete thought an ellipsis refers.

USING CONTEXT TO DETERMINE REFERENTS

Pronouns and Deletions

Once the parser has determined the existence and class (or set of classes) of a pronoun or deleted object, the context mechanism is invoked to determine the proper referent. This mechanism has a history of student interactions during the current session which contains, for each interaction, the parse (meaning) of the student's statement and the response calculated by the system. This history list provides the range of possible referents and is searched in reverse order to find an object of the proper semantic class (or one of the proper classes). To aid in the search of the history list, the context mechanism knows how each of the procedural specialists appearing in a parse uses its arguments. For example, the specialist MEASURE has a first argument which must be a quantity and a second argument which must be a part, a junction, a section, a terminal or a node. Thus when the context mechanism is looking for a referent which can either be a PART or a JUNCTION, it will look at the second argument of a call to MEASURE but not the first. Using the information about the specialists, the context mechanism looks in the present parse and then in the next most recent parse, etc. until an object from one of the specified classes is found.

The significance of using the specialist to filter the search instead of just keeping a list of previously mentioned objects is that it avoids mis-interpretations due to object-concept ambiguity. As an example, consider the following sequence from the sample dialogue in Chapter 3:

(5) What is the current thru the CC when the VC is 1.0?
(6) What is it when it is .8?

Sentence (5) will be recognized by the following rules from the semantic grammar:

```
$1) <REQUEST> := <SIMPLE/REQUEST> when <SETTING/CHANGE>
$2) <SIMPLE/REQUEST> := what is <MEASUREMENT>
$3) <MEASUREMENT> := <MEAS/QUANT> <PREP> <PART>
$4) <SETTING/CHANGE> := <CONTROL> is <CONTROL/VALUE>
$5) <CONTROL> := VC
```

with a resulting semantic form of:

```
(RESETCONTROL (STQ VC 1.0)
              (MEASURE CURRENT CC))
```

RESETCONTROL is a function whose first argument specifies a change to one of the controls and whose second argument consists of a form to be evaluated in the resulting instrument context. STQ is used to change the setting of the one of controls. The first argument to MEASURE gives the quantity to be measured. The second specifies where it is to be measured. To recognize sentence (6), the application of rules $2 and $5 are changed. There is an alternative rule for <SIMPLE/REQUEST> which looks for those anaphora which refer to a measurement. These phrases, such as "it", "that result" or "the value", are recognized by the non-terminal <MEASUREMENT/PRONOUN>. The alternative to $2 which would be used to parse (6) is:

```
<SIMPLE/REQUEST> := what is <MEASUREMENT/PRONOUN>
```

The semantics of <MEASUREMENT/PRONOUN> indicate that an entire measurement has been deleted. The alternative to rule $5:

```
<CONTROL> := it
```

recognizes "it" as an acceptable way to specify a control. The resulting semantic form for sentence (6) is:

```
(RESETCONTROL (STQ (PREF '(CONTROL)) .8)
              (PREF '(MEASUREMENT)))
```

The function PREF searches back through the context of previous semantic forms to find the most recent mention of a member one of the classes. In the above example, it will find the control VC but not CC because the character imposed on the arguments of MEASURE is that of a "part" not a

---

[7] The character imposition as described is too strong. For example:
$1) What are the specs of Q5?
$2) What is the voltage at its emitter?
The character imposed on Q5 in $1 is that of a part which means that the context mechanism invoked by $2 which is looking for a transistor won't find it. This example is handled by relaxing the restrictions the procedural specialist in $1 puts on its argument (i.e. it can be either a PART or a TRANSISTOR). In spite of this weakness in the argument limitation approach, we have found it to be a useful means of reducing the search time and avoiding some obvious mis-interpretations.

"control".[7] The presently recognized classes for deletions are PART, TRANSISTOR, FAULT, CONTROL, POT, SWITCH, DIODE, MEASUREMENT and QUANTITY. (The members of the classes are derived from the semantic network associated with a circuit.)

Referents for Ellipses

If the problem of pronoun resolution is looked upon as finding a previously mentioned object for a currently specified use, the problem of ellipsis can be thought of as finding a previously mentioned use for a currently specified object. For example,

(7) What is the base current of Q4?
(8) In Q5?

The given object is "Q5" and the earlier function is "base current". For a given elliptic phrase, the semantic grammar identifies the concept (or class of concepts) involved. In (7), since Q5 is recognized by the non-terminal <TRANSISTOR/SPEC>, the class would be TRANSISTOR. The context mechanism then searches the history list for a specialist in a previous parse which accepted the given class as an argument. When one is found, the new phrase is substituted into the proper argument position and the substituted meaning is used as the meaning of the ellipsis.

Limitations to the Context Mechanism

The method of semantic classification to determine reference is very efficient and works well over our domain. It definitely does not solve all the problems of reference. Charniak has pointed out the substantial problems of reference over a domain as seemingly simple as children's stories (1972). One of his examples demonstrates how much world knowledge may be required to determine a referent (1972 p. 7).

> Janet and Penny went to the store to get presents for Jack. Janet said "I will get Jack a top" "Don't get Jack a top" said Penny. "He has a top. He will make you take it back."

Charniak argues that to understand to which of the two tops "it" refers, requires knowing about presents, stores and what they will take back, etc. Even in domains where it may be possible to capture all of the necessary knowledge, classification may still lead to ambiguities. For example, consider the following:

(9) What is the voltage at Node 5 if the load is 100?
(10) Node 6?
(11) 7?

In (11) the user means Node 7. In (10), he has reinforced the use of ellipsis as referring to node number. (For example, leaving out (10), sentence (11) is much more awkward.) On the other hand if (11) had been "1000" or if (10) had been "10?", things would be more problematic. When (11) is "1000", we can infer that he means a load of 1000 because there is no node 1000. If (10) had been "10?", there would be genuine ambiguity slightly favoring the interpretation as a load because that was the last number mentioned. The major limitation of the current technique, which must be overcome to tackle significantly more complicated domains, is its inability to return more than one possible referent. That is, it considers each one at a time until it finds one which is satisfactory. It cannot hold off on one to see if there are any better. At present it has no metric to measure "better". The work involved in developing such a technique has not been justified by our experience.

RELATIONSHIP TO OTHER SEMANTIC SYSTEMS

The relationship between semantic grammars and purely semantic systems (Quillian 1969) (Schank et al 1975) and to some extent Wilks (1973a, 1973b) parallels the distinction between procedural and declarative knowledge. The relationship that exists between nodes in the semantic network structure contains little or no information about how these relationships might be expressed in language. An interpretation mechanism must decide where the information is useful. While this is, in some sense, more general (the same information can be used for several purposes given the proper interpreters). it is necessarily less efficient. (Wilks has extracted some expressive information, primarily concept order, into his templates.) A semantic grammar, on the other hand, is written for the process of recognizing concepts as they are expressed in the surface structures.

FUZZINESS

Having the grammar centered around semantic categories allows the parser to be sloppy about the actual words it finds in the statement. This sense of having a concept in mind, and being willing to ignore words to find it, is the essence of keyword parsing schemes. It is effective in those cases where the words that have been skipped are either redundant or specify gradations of an idea which are not distinguished by the system. For example, in the sentence "Insert a very hard fault", "very" would be ignored which is effective because the system does not have any further structure over the class of hard faults. In the sentence, "What is the voltage across resistor R8?" resistor can be ignored because it is implied by "R8".[8]

One advantage that a procedural encoding of the grammar (discussed later) has over pattern matching schemes in the implementation of fuzziness is its ability to control exactly where words can be ignored. This provides the ability to blend pattern matching parsing of those concepts which are amenable to it with the structural parsing required by more complex concepts. The amount of fuzziness (i.e. how many (if any) words in a row can be ignored) is controlled in two ways. First, whenever a grammar rule is invoked, the calling rule has the option of limiting the number of words that can be skipped. Second, each rule can decide which of its constituent pieces or words are required and how tightly controlled the search for them should be. In SOPHIE, the normal mode of operation of the parser is tight in the beginning of a sentence but more fuzzy after it has made sense out of something.

---

[8] The first of these examples could be handled by making "very" a noise word (i.e. deleting it from all sentences). Resistor however is not a noise word in all cases (e.g. "What is the current through the current sensing resistor?") and hence cannot be deleted.

Fuzziness has two other advantages worth mentioning briefly. It reduces the size of the dictionary because all known noise words don't have to be included. In those cases where the skipped words are meaningful, the mis-understanding may provide some clues to the user which allow him to restate his query.

PREPROCESSING

Before a statement is parsed, three operations are performed on the statement by a pre-processor. The first expands abbreviations, deletes known noise words, and canonicalizes similar words to a common form. The second operation is a cursory spelling correction. The third operation is a reduction of compound words.

Spelling correction is attempted on any word of the input string which the system does not recognize. The spelling correction algorithm[9] takes the (possibly) misspelled word and a list of correctly spelled words and determines which (if any) of the correct words is close to the misspelled word (using a metric determined by number of transpositions, doubled letters, dropped letters, etc.). During the initial preprocessing, the list of correct words is very small (approximately a dozen) and is limited to very commonly misspelled words and/or words which are critical to the understanding of a sentence. The list is kept small so that the time spent attempting spelling correction, prior to attempting a parse, is kept to a minimum. Remember that the parser has the ability to ignore words in the input string so we do not want to spend a lot of time correcting a word which won't be needed in understanding the statement. But notice that certain words can be critical to the correct understanding of a statement. For example, suppose that the phrase "the base emitter current of Q3" was incorrectly typed as "the bse emitter current of Q3". If "bse" were not recognized as being "base" the parser would ignore it and (mis-)understand the phrase as "the emitter current of Q3", a perfectly acceptable but much different concept.[10] Because of this problem, words like "base", which if ignored have been found to lead to misunderstandings, are considered critical and their spelling is corrected before any parse is attempted. Note that there are a lot of words (e.g. "capacitor", "replace", "open", etc.) which if misspelled would prevent the parser from making sense of the statement but would not lead to any mis-understandings. These words are therefore not considered to be critical and would be corrected in the second attempt at spelling correction which is done after a statement fails to parse.

---

[9] The spelling correction routines are provided by INTERLISP and were developed by Teitelman for use in the DWIM facility (Teitelman 1969, 1974).

Compound words are single concepts which appear in the surface structure as a fixed series of more than one word. Their reduction is very important to the efficient operation of the parser. For example, in the question "what is the voltage range switch setting?", "voltage range switch" is rewritten as the single item "VR". If not rewritten, "voltage" would be mistaken as the beginning of a measurement (as in "what is the voltage at N4") and an attempt would have to be made to parse "range switch setting" as a place to measure voltage. Of course after this failed, the correct parse can still be found, but reducing compound words helps to avoid backtracking. In addition, reduction of compound words simplifies the grammar rules by allowing them to work with larger conceptual units. In this sense, the preprocessing can be viewed as a preliminary bottom-up parse that recognizes local, multi-word concepts.

IMPLEMENTATION

Once the dependencies between semantic concepts have been expressed in the BNF form, each rule in the grammar is encoded (by hand) as a LISP procedure. This encoding process imparts to the grammar a top-down control structure, specifies the order of application of the various alternatives of each rule, and defines the process of pattern matching each rule. The resulting collection of LISP functions constitutes a goal-oriented parser in a fashion similar to SHRDLU (Winograd 1973) (but without the backtracking ability of PROGRAMMAR).

---

[10] To minimize the consequences of such mis-interpretation, the system always responds with an answer which indicates what question it is answering, rather than just giving the numeric answer.

As has been argued elsewhere (Woods 1970) (Winograd 1973), encoding the grammars as procedures (i.e. including the notion of process in the grammar) has advantages over using traditional phrase structure grammar representations. Four of these advantages are:

i) the ability to collapse common parts of a grammar rule while still maintaining the perspicuity of the grammar.

ii) the ability to collapse similar rules by passing arguments (as with SENDR).

iii) the ease of interfacing other types of knowledge (in SOPHIE, primarily the semantic network) into the parsing process.

iv) the ability to build and save arbitrary structures during the parsing process.[11]

In addition to the advantages it shares with other procedural representations, the LISP encoding has the computational advantage of being compilable directly into efficient machine code. The LISP implementation is efficient because the notion of process it contains (one process doing recursive descent) is close to that supported by physical machines, while those of ATNs and PROGRAMMAR are non-deterministic and hence not directly translatable into present architecture. (In Chapter 6 we shall see how it is possible to minimize this mismatch.) Appendix B describes the details of the LISP implementation and provides an example of a rule from the grammar.

In terms of efficiency, the LISP implementation of the semantic grammar succeeds admirably. The grammar written in INTERLISP (Teitelman 74) can be block compiled. Using this technique, the complete parser takes about 5K of storage and parses a typical student statement consisting of 8 to 12 words in around 150 milliseconds! Appendix C presents parses and timings of some of the sentences used in the dialogue.

---

[11] This ability is sometimes provided by allowing augments on phrase structure rules.

Chapter 5

LIMITATIONS OF THE LISP IMPLEMENTATION

Using the techniques described in Chapter 4, a natural language front-end was constructed capable of supporting the dialogue presented in Chapter 3 and requiring less than 200 milliseconds cpu time per question. In addition, these same techniques were used to build a front-end for NLS-SCHOLAR (Grignetti et al 1974) (Grignetti et al 1975) (built by C. Hausmann), and an interface to an experimental laboratory for exploring mathematics using attribute blocks (Brown et al 1975). In the construction of these varying systems, the notion of semantic grammar was found to be useful. The LISP implementation, however, was found to be a bit unwieldy. While expressing the grammar as programs has benefits in the area of efficiency and allows complete freedom to explore new extensions, the technique is lacking in perspicuity. The lack of perspicuity has three major drawbacks. (1) One is the difficulty encountered when trying to modify or extend the grammar. (2) The second is the problem of trying to communicate the extent of the grammar to either a user or a colleague. (3) The third is the problem of trying to re-implement the grammar on a machine which does not support LISP. These difficulties have been partially overcome by using a second, parallel representation of the grammar in a BNF-like specification language. (This is the representation which we have been presenting throughout this report.) This, however, requires supporting two different representations of the same information and does not really solve problems (1) or (3). The solution to this problem is a better formalism for expressing (and thinking about) semantic grammars.

The ATN formalism was seriously considered at the beginning of the SOPHIE project, but rejected as being too slow. In the course of developing the LISP grammar, it became clear that the primary reason for a significant different in speed between an ATN grammar and a LISP one is due to the fact that processing the ATN is an interpreted process whereas LISP is compilable. The next chapter describes an ATN compiling system which was developed to speed up the ATN approach. In this chapter we will discuss the advantages we hoped to gain by using the ATN formalism.

The advantages of using the ATN formalism fall into three general areas: (1) conciseness, (2) conceptual effectiveness and (3) available facilities. By conciseness we mean that writing a grammar as an ATN takes less characters than LISP. The ATN formalism gains conciseness by not requiring the specification of details in the parsing process at the same level required in LISP. Most of these differences stem from the fact that the ATN assumes it has a machine whose operations are designed for parsing, while LISP assumes it has a lambda calculus machine. For example, a lambda calculus machine assumes a function has one value. A function call to look for an occurrence at a non-terminal while parsing (in ATN formalism a PUSH) must return at least two values: the structure of the constituent found and the place in the input where the parsing stopped. A good deal of complexity is added to the LISP rules to maintain the free variable which has to be introduced to return the structure of the constituent. Other examples of unnecessary details include the binding of local variables and the specification of control structure as ANDs, ORs and CONDs.

The conciseness of the ATN results in a grammar which is easier to change, easier to write and debug, and easier to understand (and hence to communicate). We realize that conciseness does not necessarily lead to these results (APL being a prime example in computer languages, mathematics in general being another), however this is not a problem. The correspondence between the grammar rules in LISP and ATN is very close. The concepts which were expressed as LISP code can be expressed in nearly the same way as ATNs but in fewer symbols.

The second area of improvement deals with conceptual effectiveness. Conceptual effectiveness is, informally, the degree to which a language encourages one to think about problems in the right way. One example of conceptual effectiveness can be seen by considering the implementation of

---

[1] See (Bruce 1975) for a discussion of case systems).

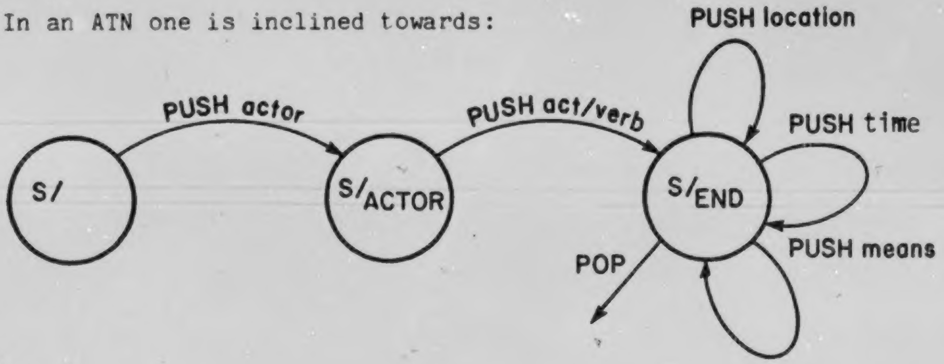
case structured rules.[1] In a typical case structure rule, the verb expresses the function or relation name and the subject, object and prepositional phrases express the arguments of the function or relation. Let us assume for the purpose of this discussion that we are looking at four different cases (agent, location, means, and time) of the verb GO (e.g. John went to the store by car at 10 o'clock). In a phrase structure rule oriented formalism one would be encouraged to write:

```
<statement> := <actor> <action/verb> <location> <means> <time>
```

Since the last three cases can appear in any order, one must also write 5 other rules:

```
<statement> := <actor> <action/verb> <location> <time> <means>
    .
    .
    .
```

In an ATN one is inclined towards:



which expresses more clearly the case structure of the rule. There is no reason why in the LISP version of the grammar one couldn't write loops which are exactly analogous to the ATN (the ATN compiler after all produces such code!) but a rule oriented formalism does not encourage one to think this way. An alternative rule implementation is:

```
  <action>:= <actor><action/verb><action1>
  <action1>:= <action1><temporal>
  <action1>:= <action1><location>
  <action1>:= <action1><means>
```

this is easier (shorter) to write but has the disadvantage of being left-recursive. To implement it, one is forced to write the LISP equivalent of the ATN which creates a difference between the rule representation and the actual implementation. This method also has the disadvantage of introducing an unmotivated non-terminal.

Another conceptual advantage of the ATN framework is that it encourages the postponing of decisions about a sentence until a differential point is reached, thereby allowing potentially different paths to stay together. In the rule oriented SOPHIE grammar there are top level rules for <set>, a command to change one of the control settings and <modify>, a command to fault the instrument in some way. Sentence (1) is a <set> and sentence (2) is a <modify>.

(1) Suppose the current control is high.
(2) Suppose the current control is shorted.

The two parse paths for these sentences should be the same for the first five words, but they are separated immediately by the rules <set> and <modify>.[2] An ATN encourages structuring the grammar so that the decision between <set> and <modify> is postponed so that the paths remain together. It could be argued that the fact that this example occurred in SOPHIE's grammar is a complaint against top-down parsing or semantic grammars or just our particular instantiation of a semantic grammar. We suspect the latter but argue that rule representations encourages this type of behavior.

Another conceptual aid provided by ATNs is their method of handling ambiguity. Our LISP implementation uses a recursive descent technique (which can alternatively be viewed as allowing only one process). This requires that any decision between two choices be made correctly because there is no way to come back and try out the other choice after the decision is made. At choice points, a rule can, of course, "look ahead" and gain information on which to base the decision (similar to the "wait-and-see" strategy used in (Marcus 1975)) but there is no way to back up and remake a decision once it has returned.

---

[2] The degree to which the separation of paths is a problem can be greatly reduced using a preprocessing "compilation" stage <Klovstad 1977> which (among other things) collapses rules with the same initial parts. In our example, however this may not work since the phrase "the current control" may be parsed as the non-terminal <CONTROL> in (1) and as the non-terminal <PART> in (2). Of course this would be a poor choice of grammar rules, and no one aware of sentences (1) and (2) would handle it this way. The problem is recognizing where situations such as this occur.

The effects of this can be most easily seen by considering the lexical aspects of the parsing. A prepass collapses compound words, expands abbreviations, etc. This allows the grammar to be much simpler because it can look for units like "voltage/control" instead of have to decode the noun phrase "voltage control". Unfortunately without the ability to handle ambiguity, this rewriting can only be done on words which have no other possible meaning. So for example, when the grammar is extended to handle:

(3) Does the voltage control the current limiting section?

the compound "voltage/control" would have to be removed from the prepass rules and included in the grammar. This reduces the amount of bottom-up processing which can be done and results in a slower parse. It also makes compound rules difficult to write because all possible uses of the individual words must be considered to avoid errors. Another example is the use of the letter "C" as an abbreviation. Depending on context, it could possibly mean either current, collector or capacitor. Without allowing ambiguity in the input, it could not be allowed as an abbreviation unless recognized explicitly by the grammar.

The third general area in which ATNs have an advantage is in the available facilities to deal with complex linguistic phenomena. While our grammar has not exparded to the point of requiring any of the facilities yet, the availability of such facilities cannot be ignored as an argument favoring one approach over another. A primary example is the general mechanism for dealing with coordination in English described in (Woods 1973a).

Chapter 6

A COMPILING SYSTEM FOR AUGMENTED TRANSITION NETWORKS*

INTRODUCTION

The augmented transition network (ATN) formalism was developed as a conceptually and computationally efficient representation for natural language grammars, and has been used successfully in several natural language processing systems (Woods et al 1972), (Simmons 1973), and (Bates 1975). Its advantages over other formalisms have been argued elsewhere (Woods 1969, 1970) and can be characterized as (1) perspicuity (2) generative power, (3) efficiency of representation, (4) flexibility for experimentation and (5) efficiency of operation.

In all of the above natural language systems, the ATN grammar has been viewed as a data structure which is interpreted by a program (called a parser). The LUNAR parser (Woods 1973a) provides a good example of such an interpretive program. This paper describes a system which views the augmented transition network as a virtual machine description and compiles it into a program executable on a physical machine, thereby eliminating the "parser".[1]

The major significance of compiling an ATN is the dramatic reduction in the amount of time required to process a sentence. The reduction is sufficient to challenge the view that a general scheme for natural language processing is too impractical to be seriously considered in the design of information management, CAI, and numerous other kinds of computer systems. Results indicate that the programs produced by the compiler parse sentences about 10 times faster than the LUNAR parser (using the LUNAR grammar (Woods et al 1972)). Also significant is that the ATN compiling system provides a testbed to explore the trade-off between the benefits of features in the abstract ATN machine and their implementation on a particular physical machine.

---

[1] This is also the viewpoint taken by Kaplan (1973) and Kay (1973), who have developed a somewhat different ATN compiler and with whom we have shared ideas in the course of this work.

*This chapter is a revised version of an earlier paper written with William Woods and presented at the International Conferences on Computational Linguistics, Ottawa, Canada, 1976.

AUGMENTED TRANSITION NETWORKS

Some years ago, Chomsky (1957) introduced the notion that the processes of language generation and language recognition could be viewed in terms of a machine. One of the simplest of such models (machines) is the finite state machine. This machine starts off in its initial state looking at the first symbol (or word) of its input sentence and then moves from state to state as it gobbles up the remaining input symbols. The sentence is accepted if the machine stops in one of its final states after having processed the entire input string; it is rejected otherwise. A convenient way of representing a finite state machine is as a transition graph, in which the states correspond to the nodes of the graph and the transitions between states correspond to its arcs. Each arc is labelled with a symbol whose appearance in the input can cause the given transition.

In an augmented transition network the notion of a transition graph has been modified in three ways. One is the addition of a recursion mechanism which allows the labels on the arcs to be non-terminal symbols which themselves correspond to networks. The second is the addition of arbitrary conditions on the arcs which must be satisfied in order for an arc to be followed. The third is the inclusion of a set of structure building actions on the arcs, together with a set of registers for holding partially built structures.[2] Figure 6.1 is a specification of a language for representing augmented transition networks. The specification is given in the form of an extended context free grammar in which alternative ways of forming a constituent are represented on separate lines and the symbol "+" is used to indicate arbitrarily repeatable constituents.[3] The non-terminal symbols are lower case English descriptions enclosed in angle brackets. All other symbols except "+" are terminals. Non-terminals not given in Figure 6.1 have names which should be self-explanatory.

---

[2] This discussion follows closely a similar discussion in Woods (1970) to which the reader is referred. If the reader is familiar with the ATN formalism he/she may wish to skip to the section "Developments to the ATN Formalism".

[3] "+" is used to mean 0 or more occurrences. While the accepted usage of "+" is 1 or more, the accepted symbol for 0 or more, "*", has not been used to avoid confusion with the use of the symbol * in the ATN formalism.

FIGURE 6.1
A Language for Representing ATNs

```
<transition network> := (<arc set> <arc set>+)
<arc set> := (<state> <arc>+)
<arc> := (CAT <category name> <test> <action>+ <term act>)
         (WRD <word> <test> <action>+ <term act>)
         (PUSH <state> <test> <action>+ <term act>)
         (TST <arbitrary label> <test> <action>+ <term act>)
         (POP <form> <test>)
         (VIR <constituent name> <test> <action>+ <term act>)
         (JUMP <state> <test> <action>+)
<action> := (SETR <register> <form>)
            (SENDR <register> <form>)
            (LIFTR <register> <form>)
            (HOLD <constituent name> <form>)
            (SETF <feature> <form>)
<term act> := (TO <state>)
<form> := (GETR <register>)
          LEX
          *
          (GETF <form> <feature>)
          (BUILDQ <fragment> <register>+)
          (LIST <form>+)
          (APPEND <form> <form>)
          (QUOTE <arbitrary structure>)
```

The first element of each arc is a word indicating the type of arc. For CAT, WRD and PUSH arcs, the arc type together with the second element correspond to the label on an arc of a state transition graph. The third element is an additional test. A CAT arc can be followed if the current input symbol is a member of the lexical category named on the arc (and if the test on the arc is satisfied). A PUSH arc causes a recursive invocation of a lower level network beginning at the state indicated (if the test is satisfied). The WRD arc can be followed if the current input symbol is the word named on the arc (and if the test is satisfied). The TST arc can be followed if the test is satisfied (the label is ignored). The VIR arc (virtual arc) can be followed if a constituent of the named type has been placed on the hold list by a previous HOLD action (and the constituent satisfies the test). In all of these arcs, the actions are structure building actions and the terminal action specifies the state to which control is passed as a result of the transition. After CAT, WRD and

TST arcs, the input is advanced; after VIR and PUSH arcs it is not. The JUMP arc can be followed whenever its test is satisfied; control being passed to the state specified in the second element of the arc without advancing the input. The POP arc indicates the conditions under which the state is to be considered a final state and the form of the constituent to be returned.

The actions, forms and tests on an arc may be arbitrary functions of the register contents. Figure 6.1 presents a useful set which illustrates major features of the ATN. The first three actions specified in Figure 6.1 cause the contents of the indicated register to be set to the value of the indicated form. SETR causes this to be done at the current level of computation, SENDR at the next lower level of embedding (so that information can be sent down during a PUSH) and LIFTR at the next higher level of computation (so that additional information can be returned to higher levels). The HOLD action places a form on the HOLD list to be used at a later place in the computation by a VIR arc. SETF provides a means of setting a feature of the constituent being built.

GETR is a function whose value is the contents of the named register. LEX is a form whose value is the current input symbol. * is a form whose value depends on the context of its use: (1) in the actions of a CAT arc, the value of * is the root form of the current input word; (2) in the actions of a PUSH arc, it is the value of the lower computation; and (3) in the actions following a VIR arc, the value of it is the constituent removed from the HOLD list. GETF is a function which determines the value of a specified feature of the indicated form (which is usually *). BUILDQ is a general structure building form which places the values of the given registers into a specified tree fragment. Specifically, it replaces each occurrence of + in the tree fragment with the contents of one of the registers (the first register replacing the first occurrence of +, the second register the second, etc.). In addition, BUILDQ replaces occurrences of * by the value of the form *. The remaining three forms make a list out of the specified arguments (LIST), append two lists together to make a single list (APPEND) and produce as a value the (unevaluated) argument form (QUOTE). A sample augmented transition network is given in Appendix D.

Developments to the ATN Formalism

A new version of any system affords an opportunity for redesign, allowing one both to overcome noticed shortcomings and to build in handles for possible future developments. In this section we will describe the differences which have evolved between the ATN formalism described in Woods' original paper (1970) and that used by the compiling system. Several of the conventions were developed before work on the compiler was begun and we shall note the first occurrence of each. This discussion is intended to bring these modifications together in one place.

The modifications were relatively minor and indicate the strength of the ATN formalism as a language for expressing grammars. The changes made were with the intention of aiding the grammar writer in building more structured grammars. One change was the addition of "feature" registers. Many tests of grammaticality rely on feature information associated with structured constituents: for example, the test of person-number agreement requires feature information from both the subject noun phrase and the verb. In the original formalism, features were only recognized as being associated with words, the structure associated with constituents had to have feature information built into it. This caused two undesirable results: tests which used the features had to look inside structures to find information and the job of changing structures was made fairly difficult. Feature registers provide the means to separate the information-saving role from the structure-saving role, resulting in cleaner, more efficient grammars.

Another difference is the inclusion of the JUMP arc type and the removal of the JUMP termination action.[4] The JUMP termination action in the original formalism allowed an arc to be taken without advancing the input. Since POP, PUSH and VIR arcs never advance the input, to decide whether or not an arc advanced the input required knowledge of both the arc type and termination action. The introduction of the JUMP arc (which is equivalent to a TST arc with a JUMP termination action in the old format) means that the input advancement is a function of the arc type alone. This is a minor conceptual improvement which simplifies programs which use the grammar to create, for example, cross references or "grammar indexes" (Bates 1975).

---

[4] The JUMP arc was originally included in the LUNAR system (Woods et al 1972).

The third difference comes from extending WRD arcs [5] to check for either: (1) a given single word; (2) an explicit list of words, thereby subsuming the MEM Arc; or (3) a list of words which is the value of a given variable. The latter feature is mainly useful as a simple method of exploring new categories without modifying the dictionary.

THE GENERAL NOTION OF ATN COMPILATION

The "compiling" of an augmented transition network grammar refers to the process of translating the ATN into machine runnable language instructions. The ATN is a description of "what sentences the machine should accept" while the compiled ATN must additionally be a description of "how the machine should accept them". The compiling process requires decisions about characteristics of the parsing process which are left unspecified by the ATN formalism. The additional decisions needed by the ATN compiler fall roughly into two classes. The first is what constitutes a configuration of the ATN machine. A configuration is the amount of information needed to completely characterize the status of the machine at the moment in time on one of its computation paths during the processing of a sentence.[6] For example, a configuration of a simple finite-state network consists of only the name of the state of the machine and input string remaining to be parsed. A configuration for the ATN used in the LUNAR system (Woods et al 1972) needs a state, a string pointer, a recursion stack, registers, a hold list and a path. The second kind of decision unspecified by the ATN formalism is the control structure that the ATN machine is to have. That is, in what order should the alternative paths through the grammar be tried, e.g. should the search strategy be depth-first or breadth-first. These details must be added to the details specified in the ATN in order to define a machine (program) in a runnable computer language.

---

[5] This extension of WRD arcs was developed in the BBN speech system (Bates 1975).

[6] One of the characteristics of the ATN is that the machines specified can be non-deterministic. This means that from a given configuration, there may be more than one possible next configuration (more than one of the arcs leaving a state can be followed). Since any implementation of an ATN machine on a serial computer will not be able to follow these paths simultaneously, it must have a mechanism for remembering alternative possible configurations.

Once the details about the structure of the configuration have been decided upon,[7] the compiler can translate each arc in the ATN grammar into statements in the object language which (when executed) test the conditions on that arc and if successful will carry out the desired changes to the configuration in the compiled machine. Once a control structure has been specified, the compiler can put the code from the arcs together in a way which manifests that control structure. By changing the details of these decisions the compiler can generate, from a single ATN, many different machines all of which will give the same parses for the same sentences but whose internal structure and efficiency are quite different.[8]

Areas of Optimization

Since the primary reason for compiling an otherwise interpreted process is speed, care has been taken to isolate those areas of the process which can be optimized. There are three general areas in which the ATN machines can be optimized independent of the particular computer

---

[7] Decisions about internal details are made either explicitly as a declaration to the compiler or implicitly by technical license of the designer of the compiler. This will be discussed further later.

---

[8] Those readers familiar with LISP may find it useful to refer to Appendix D which contains a simple ATN grammar together with annotated examples of the programs which were compiled from it under different specifications.

implementation. One concerns the amount of information which must be saved in a configuration (an active state of the processing of a sentence by the ATN).[9] The information required in a configuration to implement the ATN as described earlier in this chapter includes (1) the state, (2) the input, (3) the list of untried arcs, (4) the stack of higher level arcs which are waiting for completion of this level of the computation, (5) a list of registers and their contents, (6) a list of features and their contents and (7) a hold list. If, for example, a particular ATN grammar does not use the hold list (i.e. contains no VIR arcs), or if this facility is not needed for a particular application of the grammar, the hold list need not be included in the configuration. Other possible ATN mechanisms like "weight" information (see Woods (1973a) or Bates (1975)) require additional information to be associated with a configuration. If most configurations have the same weight, it is possible to implement this feature so that it is not part of every configuration (e.g. via hash links (Bobrow 1975)). The less information a configuration requires, the less storage it uses and the less time it takes to create.

The ATN machine can also be optimized with respect to the number of configurations which must be created. If the view that making an arc transition creates a new configuration is taken literally, then each arc would be compiled into code which if taken creates two new configurations. One would be created to continue at the state at the other end of the arc, and another one would be needed to remember the remaining arcs of the present state. In most cases, however, the current configuration does not need to be saved, allowing one or the other of these two new configurations to be made by destructively modifying the old one. In a depth-first strategy, the arc can be compiled so that it changes the current configuration after creating a new alternative one to examine the other arcs. In a breadth-first strategy, the arc can be translated to create a new configuration to continue processing at its tail, and the current configuration can be changed to examine the other arcs leaving the current state.

---

[9] In this discussion, we will mean by configuration the data structure which contains the information necessary to represent a state in the virtual machine. Thus the creation and modification of configurations are data structure operations.

The third major area of optimization deals with taking advantage of features of the target language to produce code which runs efficiently on a physical machine (assuming the target language runs efficiently on a physical machine). For example, in most object languages, an arc can be represented by a sequence of statements in a program associated with a label identifying it. If the arc succeeds, it can GO to the first arc of the next state. If it fails, it can "fall through" to the next arc leaving the state. In languages which allow the direct use of accumulators, it may be possible to set aside accumulators for those parts of the configuration which are accessed most often and thereby reduce the time spent accessing the configuration.

A GRAMMAR COMPILING SYSTEM

Having described the general notion of ATN compilation, we now consider the construction of a system to perform such compilations. We will describe a general compiling system which will take an ATN grammar plus user specifications of desired features, and produce an optimal compiled ATN machine. The general structure of the compiling system is shown in Figure 6.2. It consists of the following pieces:

1) An ATN grammar - provided by the user.
2) A set of user arc actions - the function definitions for arc actions which are not part of the basic ATN formalism.
3) Grammar declarations - a set of declarations to the grammar compiler which specify the control structure and features of the ATN formalism that this grammar uses and tell the compiler how to compile the user arc actions in the grammar. These declarations allow the grammar compiler to optimize the program produced from the user's grammar.
4) The grammar compiler - a program which takes the user's ATN grammar and grammar declarations and produces a LISP program. This LISP program will be referred to as the "object" code of the grammar.
5) The lexical routines - includes the dictionary, dictionary retrieval routines, the lexical analysis routines and the substitution and compound word testing routines. It may also include routines to correct spelling mistakes or recognize domain dependent words such as SOQ14.

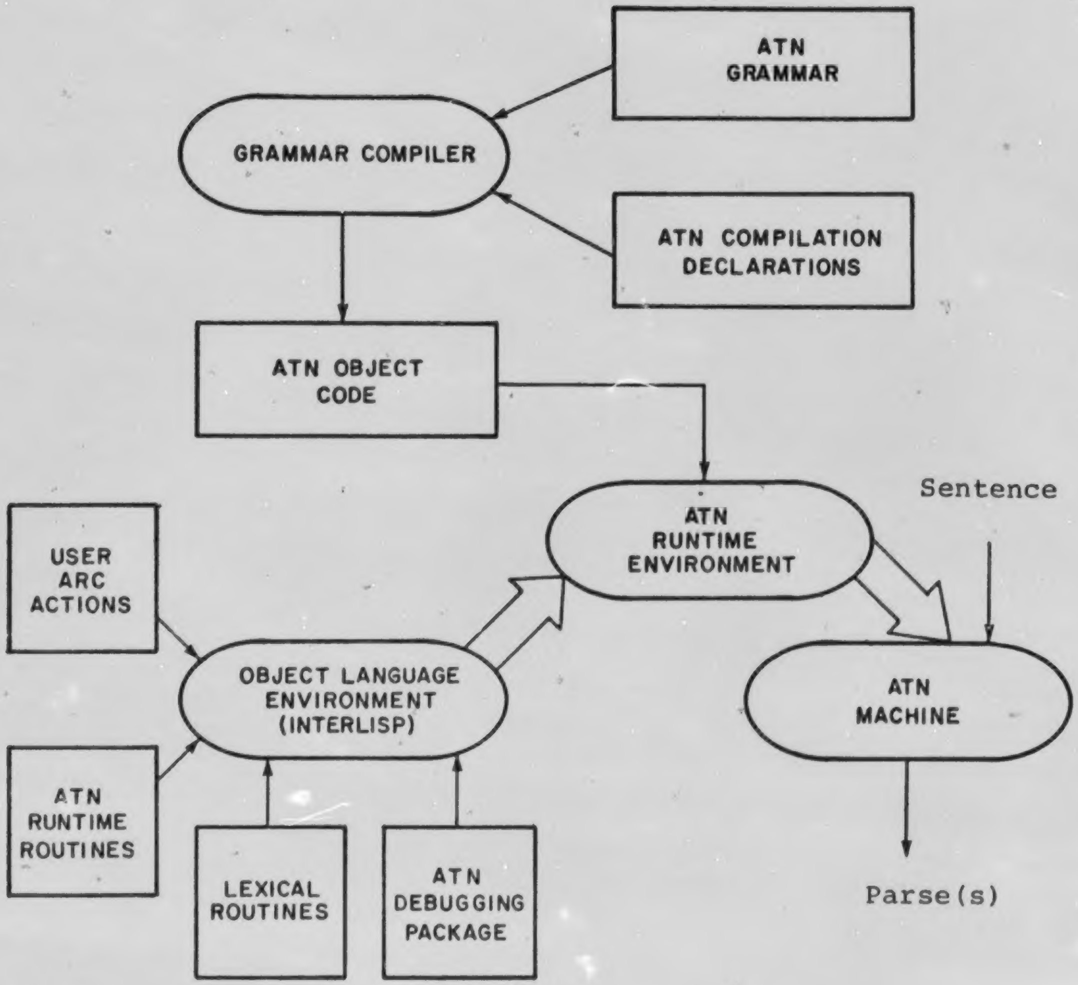

ATN GRAMMAR
GRAMMAR COMPILER
ATN COMPILATION DECLARATIONS
ATN OBJECT CODE
ATN RUNTIME ENVIRONMENT
Sentence
USER ARC ACTIONS
OBJECT LANGUAGE ENVIRONMENT (INTERLISP)
ATN MACHINE
ATN RUNTIME ROUTINES
LEXICAL ROUTINES
ATN DEBUGGING PACKAGE
Parse(s)

FIGURE 6.2

Overview of the ATN Compiling System

6) The runtime ATN functions - the functions which perform the actions and tests described in the ATN formalism (e.g. SETR, GETF, etc.) and which maintain the configurations.

7) The debugging package - a set of functions useful for tracing and debugging the grammar object code.

These pieces are used in the following manner: The ATN grammar (1) and the grammar declarations (3) are input to the grammar compiler (4) which produces the ATN grammar object code. The user's arc actions (2), the lexical routines (5), the runtime ATN functions (6), and the debugging package (7) are loaded together to create a runtime environment for the grammar object code. The grammar object code is then loaded into the runtime environment, which results in an ATN machine (i.e. a program which takes sentences as inputs and produces parses as outputs).

The central issue in the compiling system is the form of the object code produced by the grammar compiler. Once the form of the object code has been decided, the tasks of generating it and developing a runtime environment for it become straightforward. For this reason we will describe the ATN object code in some detail.

A VERSION OF THE GRAMMAR COMPILING SYSTEM

As a first step in building the general compiling system, a version was written which produced ATN machines with a particular set of features. Since then, the first version has been extended to allow choices in a number of areas, but the general implementation has not yet been completed. In this section we shall describe this first version of the ATN compiling system and the class of machines it builds. The first version will provide a concrete basis from which we can consider changes and extensions.

Features of the ATN machines

The first ATN compiler was designed to produce machines in INTERLISP[10] which had all of the facilities of the LUNAR parser except the well-formed substring table and SYSCONJ facility (Woods et al 1972). In addition to the standard arc types (CAT, WRD, TST, JUMP, PUSH, POP, and VIR) and arc actions, these facilities include RESUME, SUSPEND and ABORT actions; lexical alternatives and the accessing of registers from higher levels. The control strategy compiled into the ATN machine is the same as that normally used by the LUNAR parser, searching the paths through the grammar in a depth-first manner. As will be seen, the actual task of compiling an ATN is fairly straightforward once one knows the form the compiled ATN will have and the run-time environment in which it resides. For this reason, a larger portion of this section will concentrate on the result of the ATN compiler rather than on the compiler itself. Before we describe the structure of resulting ATN machines, we shall describe in some detail, decisions in three areas which determine aspects of the ATN machine and its run-time environment. These areas are: lexical analysis, configuration make-up and control structure.

---

[10] This object language was chosen partially because the LUNAR parser is written in INTERLISP and many of the lexical routines and arc actions could be borrowed.

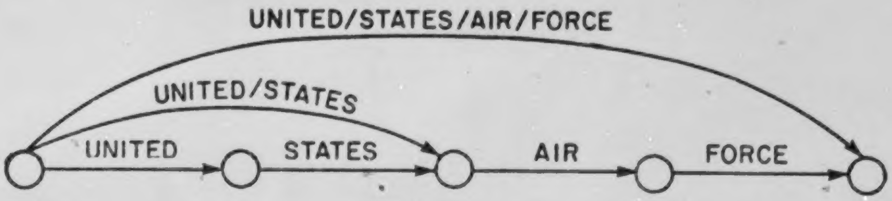
Lexical Preprocessing

In most implementations of ATN parsers, some amount of processing is done before the input is passed through the grammar. This processing includes the operations of dictionary look-up, morphological analysis, and substitution and compound word checks.[11] In the LUNAR parser, this analysis is done as the parser "needs" the next word in the input and the ambiguities which arise from any or all of these operations are handled by allowing the lexical processor to create alternative configurations. In the compiled implementation, the lexical operations are done during a prepass. To capture some of the ambiguities, the input is converted into a chart representation (Kay 1964). For example the phrase "United States Air Force" would result in the chart structure given in Figure 6.3 (assuming there were compound rules for United States and United States Air Force). In Figure 6.3, the circles represent nodes of the chart and the arrows represent edges between them. Ambiguities, which are represented in the chart as more than one edge leaving a node, result only from possible substitutions and compound words. Alternative interpretations of a word as different categories are all stored on the same edge, which preserves the ordering of arcs coming out of a state with respect to alternative categories of a word. (Multiple interpretations of a word under the same category will be discussed later.)

---

[11] Substitutions allow a word to be replaced by another word or series of words, (e.g. "can't" replaced by "can not"). Compound rules allow a series of words to be replaced by a single word (e.g. "United Stated Air Force" replaced by "United/States/Air/Force").

Figure 6.3
Example of an input chart



There are two motivations for changing the input string into a chart. One is that it recognizes the ambiguities before parsing begins. This removes the burden of lexical processing from the machine and allows the machine to be independent of any particular lexical analysis scheme. Creating the chart also means that any unknown words are identified before any time is spent processing the sentence. The other motivation for using the chart is that it may later be extended to include well-formed constituents as well as terminals (i.e. it could subsume the role of the well-formed substring table). This would allow bottom-up parsing keyed off lexical items and permit experimentation with combinations of bottom-up and top-down parsing. In fact the substitution and compound word mechanisms presently allow bottom-up parsing, but it is not very interesting because the resulting constituents must be terminal symbols.[12]

---

[12] Since the distinction between terminals and non-terminals is maintained in the grammar by distinguishing between CAT and PUSH arcs, one way to overcome this limitation is to combine CAT arcs with PUSH arcs. This new arc type would be defined as "look for an x and if one isn't found, PUSH for one".

Configurations

As mentioned earlier, the information required to completely characterize a state of the syntactic processing is called a configuration. A configuration for the class of machines generated by the initial implementation has the following parts:

CONFIGURATION number: a number unique to a configuration which is used to identify it, e.g. on alternative configuration lists, in traces or in paths.

STATE: the state of the grammar currently being examined. In a compiled grammar this is the label (index) of the code compiled from some particular arc of a state. That is, in addition to specifying the state in the ATN, it also indicates which arc of that state is under consideration.

NODE: a pointer into the input. The input is viewed as a set of edges because the lexical prepass can create alternative word substitutions or compound words.

STACK: a pointer to higher levels of ATN's which PUSHed to the current level.

REGS: a pointer to the list of registers available to this configuration. The registers are stored in a "forked stack" format which allows maximal sharing between configurations (Woods 1973a).

FEATS: a pointer to a list of feature registers.

HOLD: the hold list of as yet unassigned constituents.

A good deal of efficiency can be gained or lost creating and accessing configurations. To allow experimentation with different implementations, the code produced by the ATN compiler is written in terms of data accessing functions. In one of the possible implementations, a configuration is represented as a 3 word block out of an array (see Figure 6.4).

Figure 6.4
A Configuration

arc/state	node
stack	registers
features	hold list

One of the reasons that the compiled grammar is more efficient than an interpreted one is that the decision about what arc and edge to try next can be fixed at compile time and integrated into the grammar code. For our first version of the compiler, we chose a depth-first control structure. The reasons for this are: (1) depth first search takes advantage of the natural way provided by the ATN to order the arcs (i.e. the order in which they are given on the state) and for some applications of a natural language front-end, the first parse may be all that is required; (2) by using a well-formed substring table, a lot of the work done during an unsuccessful depth-first attempt can be used by later attempts; (3) a depth-first strategy is conceptually simple and easy to produce code for; and (4) while many systems have allowed a great variety of control structures, none has been shown to work consistently better than others.

Using a depth-first control structure, the arcs are tried one at a time in the order specified in the grammar. The first arc which succeeds from each state is taken and an alternative configuration which will try the remaining arcs in the state is pushed onto the alternatives stack. If a configuration blocks (none of the arcs leaving its state succeed), the top configuration on the alternatives stack is started. If there is any ambiguity in the input chart, the first lexical edge is applied to all of the arcs coming out of the state before any of the other lexical edges is tried. Note that the edges of the input chart are not used to represent alternative lexical interpretations of a word but are used to represent compound words and substitution expansions. The different lexical categories of a word are grouped together on the same edge and the desired category is chosen by a CAT arc. This method has the advantage that a WRD arc succeeds only once instead of once for each interpretation. It also provides a natural way for the grammar to order the interpretations of a word as say an N or a V. If the N interpretation is more likely at this

---

[13] If the user really does want the word itself to favor one interpretation over another, the substitution mechanism can be used to create an alternative edge in the chart.

state the CAT N arc is ordered ahead of the CAT V arc.[13]

It is possible that a word has more than one interpretation in the same category. For example "saw" can be either present tense as in "saw off the right corner" or past tense as in "I saw the man in the park". This type of ambiguity is handled by allowing the CAT arc to create an alternative configuration (called a CATALT) which will try the other possible interpretations of the word within the same category. This CATALT will be attempted before the alternative which tries the other arcs in the state.

An Overview of an ATN Machine

The task of the grammar compiler is to transform an ATN grammar into a runnable program. This program realizes one of the machines specified by the grammar, parsing sentences in a way dictated by the grammar under a particular control structure. Some of the basic operations of this machine are choosing an alternative configuration from the list of those pending, activating the configuration, and exploring arcs which leave the state of the active configuration. This section is meant to familiarize the reader with the form of a compiled ATN, by analyzing it in terms of these basic operations.

Figure 6.5 presents a flow chart of the logical operations of the ATN machine. The abstraction does not directly reflect the program structure of the compiled ATN (as will become apparent in the next section which examines an actual machine); however, it should act as a "useful myth" toward understanding the operations required of the object code. When called, the machine combines its input (a chart created by the lexical prepass described earlier) with a STATE which is the starting state of the grammar (e.g. S/) to create the initial configuration. (The other parts of the configuration are empty.) The next step is to set up the configuration (i.e. put the machine into the configuration). This operation may be viewed as getting a configuration "ready to run." When the configuration is started, it tests the arc condition on the first arc. If

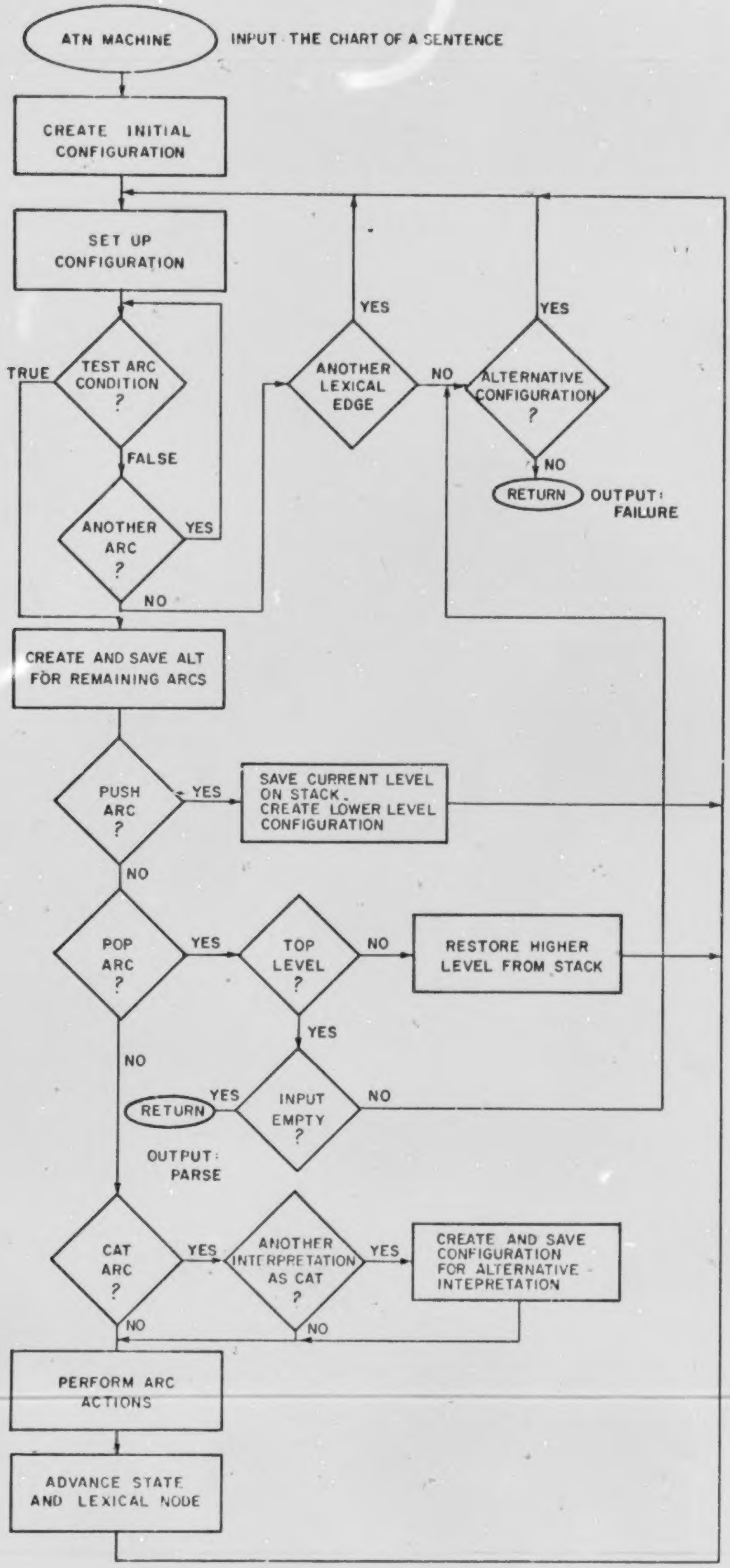

ATN MACHINE
INPUT: THE CHART OF A SENTENCE
CREATE INITIAL CONFIGURATION
SET UP CONFIGURATION
TEST ARC CONDITION ?
TRUE
FALSE
ANOTHER ARC ?
YES
NO
ANOTHER LEXICAL EDGE
YES
NO
ALTERNATIVE CONFIGURATION ?
YES
NO
RETURN
OUTPUT: FAILURE
CREATE AND SAVE ALT FOR REMAINING ARCS
PUSH ARC ?
YES
SAVE CURRENT LEVEL ON STACK. CREATE LOWER LEVEL CONFIGURATION
NO
POP ARC ?
YES
TOP LEVEL ?
NO
RESTORE HIGHER LEVEL FROM STACK
YES
INPUT EMPTY ?
YES
RETURN
NO
OUTPUT: PARSE
NO
CAT ARC ?
YES
ANOTHER INTERPRETATION AS CAT ?
YES
CREATE AND SAVE CONFIGURATION FOR ALTERNATIVE INTEPRETATION
NO
NO
PERFORM ARC ACTIONS
ADVANCE STATE AND LEXICAL NODE

FIGURE 6.5

Logical Control Flow of a Depth First ATN Machine the condition fails, the next arc is tried. If there are no other arcs, the next lexical interpretation (i.e. next edge of the input chart) is tried. If there are no other edges, an alternative configuration must be selected. In the depth-first scheme, the list of alternatives is maintained as a stack and the top configuration is used. If the alternatives list is empty, the input failed to parse.

If the condition on the arc succeeds, an alternative configuration is created which will try the remaining arcs of the present state, and special actions are performed dependent upon the arc type. For PUSH arcs, the current level of processing is saved in a configuration and stored on the stack. A configuration is then created beginning at the state being pushed for. For POP arcs, the configuration on the top of the stack is restored. If the stack is empty and the input chart is empty, the sentence has successfully parsed. If the input is not empty, this path has failed and an alternative is tried. For CAT arcs, if the chart edge currently under consideration has an alternative interpretation under the same category, a configuration is saved which will try the other interpretation. Regardless of the arc type, the actions on the arc being taken are performed and the next configuration is created by changing the state and/or advancing the chart. We realize that this description leaves many critical details unspecified. These details are addressed in the next section.

Anatomy of an ATN machine

A compiled ATN has the following form:[14]

```
(LAMBDA (ACF)
   (PROG (special variable, etc.)
     SPREAD-ACF (code to set up ACF, the current configuration)
                (GO EVAL-ARC)
     NEXTLEX    (if (another lex?) then (advance NODE) (GO EVAL-ARC))
     DETOUR     (if (another alt?) then (ACF←alt) (GO SPREAD-ACF))
                      else (RETURN failure))
     EVAL-ARC   (BRANCH STATE arclabel1 arclabel2 ... arclabeln)
     arclabel1  arc1 code
     arclabel2  arc2 code
     ...
     arclabeln  arclabeln code))
```

---

[14] For this implementation, INTERLISP is used as the object language for the compiler. All of the examples given here will be in CLISP (Teitelman 1975), an ALGOL-like dialect of INTERLISP. In LISP, a function call is expressed in Cambridge-Polish notation: as a parenthesized list of the function name followed by its arguments.

It is a LISP function of one argument, ACF, which is the Active ConFiguration.[15] The code by the label SPREAD-ACF sets the proper variables (by variable assignments from the fields of ACF) to establish an active configuration, ending with a jump to EVAL-ARC. For this implementation, the parts of the active configuration are stored in the variables STATE, NODE, STACK, REGS, FEATS and HOLD. The BRANCH statement associated with EVAL-ARC will cause the code associated with the STATE of the active configuration to be executed (i.e. by passing control to the label indicated by the value of STATE). The arc code portion of the machine will be described in the next section. The code by the label NEXTLEX checks to see if there is another lexical alternative in the current configuration (i.e. another edge on NODE). If there is, it changes the NODE of configuration to the next interpretation and jumps to EVAL-ARC. The code by the label DETOUR picks an alternative from the list of alternatives and jumps to SPREAD-ACF to establish it as the active configuration. If there are no more alternatives, the machine returns failure. The pieces of the ATN which we have described so far are independent of the particular ATN grammar being compiled. The bulk of the ATN machine consists of the separate pieces of code which have been compiled, one piece from each arc in the grammar. The following section describes this code.

CODE FOR THE ARCS

An arc in the grammar has three parts: (1) conditions which must be satisfied in order for the arc to be taken; (2) actions which are to be performed if the arc is taken; and (3) a termination action which moves to a new state in the grammar. The code compiled from each arc[16] first checks the conditions required by that arc. If the conditions are met, an alternative configuration is created to try the remaining arcs in the state and the code corresponding to the actions on the arc is executed. The last

---

[15] The initial configuration (which has a STATE which is the starting state of the grammar (e.g. S/) and a NODE which is the first node of the input chart) is created before this function is invoked.

---

[16] This code is similar in many ways to General Syntactic Processor code of Ronald Kaplan (1973) and Martin Kay (1973).

action of the arc will be the termination action which jumps to the first arc of the next state.

If the tests for an arc are not met, the next arc needs to be tried (or if there are no more arcs, an alternative needs to be chosen). This is implemented by allowing control to "fall through" to the next arc. At the end of the code for all of the arcs in a state, control is sent to the label DETOUR which starts an alternative.[17]

The form of code produced from each arc in the grammar is determined by its arc type. In this implementation we have allowed for seven different types: WRD, CAT, JUMP, VIR, TST, PUSH, and POP. (As mentioned earlier, the MEM arc type used in the LUNAR grammar has been subsumed under a more comprehensive WRD arc.) In the following sections, we will describe briefly the code produced from WRD arcs, CAT arcs, PUSH arcs, and POP arcs. The code produced for the TST arcs, JUMP arcs and VIR arcs is similar to WRD arcs (differing only in the test which is implied by the arc type and whether or not the terminating action advances the input). Each section is prefaced with the given arc type as it appears in an uncompiled ATN grammar, so that its relationship to the resulting compiled code is made clear.[18]

WRD arcs

```
(WRD <word> <tst> <action>+ (TO <state>))
```

The WRD arc provides a means of testing for a particular word. <Word> can be (1) a word, (2) a list of words (in which case this is similar to the LUNAR MEM arc), or (3) a variable whose value is a list of words (in the current implementation, a variable is distinguished from a word by requiring that it begins with a "/", e.g. /MONTH/). Depending on the instantiation of <word>, the test condition of the WRD arc compiles into either (1) ARCWRD, (2) ARCMEM or (3) ARCMEME, all of which check the word on the current edge of the input chart. This implied test is then embedded within the explicit test <tst> and made the test condition of a conditional statement. The consequence clause of the conditional statement is made up of the compiled versions of <action>+ and (TO <state>). For example, the first arc of the grammar state Q4/ in the sample grammar from Appendix D:

---

[17] If there are lexical alternatives, control will be sent to the label NEXTLEX which will advance to the next lexical alternative.

[18] The examples have been taken from the ATN grammar in Appendix D. Appendix D also presents the complete program which results from its compilation.

```
[Q4/
    (WRD BY (GETR AGFLAG) (SETR AGFLAG NIL) (TO Q7/))]
```

compiles into:

```
Q4/    (if (ARCWRD BY) and (GETR AGFLAG)
          then (ALTARC Q4/-2)
               (SETR AGFLAG NIL)
               (DOTO Q7/)
               (GO Q7/))
Q4/-2  ...
```

If the word test (ARCWRD BY) and the arc test (GETR AGFLAG) are both successful, the function ALTARC sets up an alternative configuration to try the other arcs of this state (beginning at Q4/-2)[19] and the actions on the arc are executed. After the setting of the AGFLAG register, the function DOTO changes the state to the next state Q7/ and advances the input. The function GO jumps to the label Q7/ which begins applying the first arc of the state Q7/. If either of the tests on this arc fail, control "falls through" to the code for the second arc Q4/-2.

CAT Arcs

```
(CAT <category> <test> <actions>+ (TO <state>))
```

---

[19] ALTARC is the primary means of creating alternative configurations. It creates, and saves on the alternatives list, a copy of the current configuration which has had its arc/state changed. The new arc/state is the argument passed to ALTARC. The two functions, ALTCAT and NEXTLEXALT, which also create configurations will be discussed later.

The CAT arc type provides a means of restricting an arc to words of a particular syntactic category. <Category> is the syntactic category being tested for. In this implementation, the set of syntactic classes to which a word belongs is stored with its dictionary entry and is included in the chart by the prepass. In addition the dictionary includes an ASSOC list of lexical features of the input word under this interpretation.[20]

The code produced for a CAT arc is complicated by the need to generate CATALTs (alternative interpretations of a word in the same category). In particular, CATALTs require creating a second arc/state (label) for each CAT arc which will process all but the first interpretation. This second arc/state is the same as the first CAT arc without the action which creates the next arc alternative (so that the next arcs will not be tried again for each of the other interpretations). For example the first arc from state VP/ in the Appendix D grammar:

```
[VP/
    (CAT V (GETF * UNTENSED) (SETR V *) (TO Q3/)]
```

compiles as:

```
VP/            (if (NOT (ARCCAT V)) then (GO VP/-2))
               (ALTARC VP/-2)
VP/-1-CONT     (ALTCAT VP/-1-CAT)
               (if (NOT (GETF * 'UNTENSED)) then (GO DETOUR))
               (SETR V *)
               (DOTO Q3/)
               (GO Q3/)
VP/-2          ...

VP/-1-CAT      (ARCCAT V)
               (GO VP/-1-CONT)
```

The interpretation of the code is as follows. If the current word is not a V, immediately try (the code compiled from) the next arc (VP/-2). Otherwise create an alternative configuration which will try the next arc (ALTARC). ALTCAT checks for other interpretations of the input word as a V and if there are any, creates an alternative with an arc/state which redoes only this arc (VP/-1-CAT).[21] If the test on the arc (GETF * 'UNTENSED) is not true, DETOUR will start the next alternative which will be either the next interpretation (the configuration created by ALTCAT) if there is one or the next arc (the one created by ALTARC).If the test on the arc is true, the arc actions are executed, the input is advanced (DOTO), and the next state is processed (GO Q3/).

---

[20] The LUNAR grammar also supports a different notion called features which is a list of atoms contained in the dictionary entry for each word under the property FEATURES and which is global to all interpretations.

The code beginning at the label VP/-1-CAT performs an ARCCAT test (which sets up *) and rejoins the first arc immediately after the ALTARC (label VP/-1-CONT). By rejoining here, the alternative which tries the next arc in this state does not get created a second time; however, the alternative to try the third (or more) alternative interpretations does get created (as it should).

PUSH Arcs

```
(PUSH <state> <test> <action>+ (TO <state>))
```

The PUSH arc provides a way of recursively invoking the grammar to find a complex constituent. The <test> condition is checked before the PUSH. The PUSH arc gets compiled into two arc/states: one which performs the pre-actions [22] and does the PUSH; and one which is returned to after the POP by the lower network, which does the arc actions and moves to the next state. For example the third arc of state Q3/:

---

[21] The phenomenon of multiple interpretations within the same category is rare and has only occurred in applications for verbs (e.g. "put" which can be either present tense as in "put the book on the table", or a past participle "the book was put on the table"). (In much more general applications it may arise in noun plurals such as "axes" or "bases.") Since its implementation so greatly complicates the code generated for a CAT arc (without it, a CAT arc would be similar to the WRD arc described earlier), the compiler allows the user to specify which categories can have multiple interpretations. For example, a CAT ADJ arc may be compiled to execute more efficiently than a CAT V arc.

[22] SENDRs, SUSPENDs and any actions which are embedded in "!" as in (! (FOO)) will be done before the push.

```
[Q3/ ...
    (PUSH NP/ (TRANS (GETR V))
              (SETR OBJ *)
              (TO Q4/))]
```

compiles into:

```
Q3/-3       (if (TRANS (GETR V))
              then (ALTARC Q3/-4)
                   (DOPUSH NP/ Q3/-3-PUSH)
                   (GO NP/))
Q3/-4       ...

Q3/-3-PUSH (SETR 'OBJ *)
            (DOPTO Q4/)
            (GO Q4/)
```

If the test (TRANS (GETR V)) is not true, the next arc is tried immediately. If the test is true, ALTARC creates and saves an alternative configuration to try the other arcs beginning at Q3/-4. The function DOPUSH performs the operations necessary to recursively call the lower level. These consist of saving the currently active configuration on the stack (after changing its state to restart at the return arc/state Q3/-3-PUSH), and then changing the state to the lower level (NP/) and changing the registers and features to those being sent down. Following the DOPUSH, the lower state is started immediately (GO NP/). The code by Q3/-3-PUSH is the return arc/state which will be executed when (if) the lower network (NP/) finishes. The structure returned by the lower network, *, is saved in the OBJ register. The function DOPTO changes the state but does not advance the input. The GO begins processing at the state Q4/.

POP Arcs

(POP <form> <test>)

The POP arc identifies a final state in a network and specifies the structure that is to be returned. The state NP/3:

```
[NP/3
    (POP (BUILDQ (NP (NPR +)) NPR) T)]
```

compiles into:

```
NP/3  (NEXTLEXALT NP/3)
      (DOPOP (BUILDQ (NP (NPR +)) NPR))
      (GO EVAL-ARC)
```

The function DOPOP re-establishes the next higher level by restoring the state and the registers from the configuration on the top of the stack. DOPOP also sets * to the proper structure. After DOPOP has established the higher level configuration, EVAL-ARC will activate it.

There are two other points exemplified by the code for NP/3. One is that since the test on the arc is always true, there is no need for a conditional statement. The other is that since the arc is the last arc in the state, it does not need to create an alternative to examine the remaining arcs. What the last arc does need to do however, is to check to see if there is an alternative lexical interpretation (e.g. try "united states" as two words after the compound "United/States" fails as in the sentence "He united states to create a smaller grammar"). This is done by the function NEXTLEXALT. If there is another lexical alternative (another edge on the input chart) NEXTLEXALT creates an alternative configuration which has the next lexical alternative as an input and has an arc/state which restarts it at the first arc of the current grammar state (NP/3).

Special Actions

Certain of the actions allowed by the ATN formalism affect the form of the compiled code. These are the actions RESUME and SUSPEND which change the standard control flow. SUSPEND provides a means of setting the weight (likelihood of success) of a configuration. When the weight of a configuration is changed, it temporarily stops in favor of other configurations which have a better weight. This is done by putting the current configuration and its weight on a list of suspended configurations and then aborting. The alternative configuration with the best weight (possibly this one again) will then be chosen. RESUME allows a lower level network which has already popped to resume processing input words from a later place in the input (for conveniently handling certain extraposition phenomena in natural English). This is implemented by allowing RESUME to simulate a PUSH. Both SUSPEND and RESUME require the creation of a new state (label) in the object program.

Compiling the Compiled ATN

The result of compiling an ATN is a program in a computer language; in our examples, a LISP program. This program can be executed by a LISP interpreter or, more likely (since the primary reason to compile the ATN is efficiency), compiled by a LISP compiler to produce machine level code. Major gains in efficiency can be made through judicious use of compiler macros and hand coding of oft used functions.[23] For example, the arc action BUILDQ which fills a fixed template with specified register contents can be expanded via macros at compile time to be direct calls on the primitive functions CONS, LIST, etc. Another good example is the use of machine language macros for the functions which access the parts of a configuration (see Figure 6.4). The indexing address mode of the hardware can then be used to access any part of a configuration in a single machine instruction. Implementing the machine language definitions as macros does not effect the debugging facilities which can use the normal LISP definitions. While techniques at this level are machine dependent (and therefore interfere with the transferability of the program), the functional nature of the object code delimits the range of this dependence.

Results

For purposes of comparing the ATN compiling system with the LUNAR parser, the ATN grammar from LUNAR was compiled using the compiling system. The control structure of the LUNAR parser and the compiled version are the same (depth-first) so that traces could be compared to ensure that both systems were performing the same task. The ATN machine was then block compiled (Teitelman 1975) and used to parse some of the example sentences taken from Woods (1973b) and the appendix of Woods et al (1972). (In this initial version of the ATN compiler, the LUNAR conjunction-handling facility SYSCONJ is not supported. However, we plan to include it, or some modified version of it, in future implementations.) All of the parses were run on a PDP KA10 which has a cycle time of approximately two microseconds. The times listed for the compiled version are "lexical prepass" + "parsing". As can be seen, the compiled version required less than one-tenth the time of the LUNAR version.

---

[23] The examples presented deal with the INTERLISP compiler, however, similar compiling features are available in many other versions of LISP.

1) Give me all analyses of S10046
LUNAR - 3.05 seconds
Compiled - .045 + .200 = .245 seconds

2) How many breccias contain olivine?
LUNAR - 1.65 seconds
Compiled - .050 + .125 = .175 seconds

3) What are they?
LUNAR - 2.25 seconds
Compiled - .030 + .150 = .180 seconds

4) List modal plag analysis for lunar samples that contain olivine.
LUNAR - 2.9 seconds
Compiled - .065 + .200 = .265 seconds

5) What is the average composition of olivine
LUNAR - 3.1 seconds
Compiled - .040 + .235 = .275 seconds

6) References on tritium production
LUNAR - 1.6 seconds
Compiled - .030 + .095 = .125 seconds

7) How many breccias do not contain Europium
LUNAR - 2.5 seconds
Compiled - .075 + .165 = .240 seconds

EXTENSIONS TO THE INITIAL VERSION

Since the initial implementation of the ATN compiler, many of the features of the ATN have been made optional. That is, the user can specify the features he needs and the compiler will produce programs optimized to those features. This allows a user to build a natural language front-end incrementally, adding new features as his subset of the language becomes sophisticated enough to use them. If the user wants to start with a finite-state, keyword ATN, the compiler will produce a very fast finite state machine. If his ATN develops to the point that he needs recursive networks, alternative lexical interpretations and a mechanism for remembering well-formed constituents, the compiler will produce a machine which has these capabilities. The point is that the user can work through the entire development in the same formalism using "upward compatible" grammars and has the opportunity to evaluate the benefits and the costs of each new addition. This also provides a metric when trying to decide between two different methods of handling the same construct. In the next section, we will describe one of the major options which has been implemented, and in the subsequent section describe optional control structures which have not been implemented. Appendix E provides a complete specification of the options which are available in the present implementation.

The Well-Formed Substring Table

The Well-Formed Substring Table (WFST) allows some of the work performed by the first attempt to parse a sentence to be re-used by the later attempts. It does this by keeping track of the complex constituents which are found as a result of PUSH arcs. Care must be taken to ensure that a PUSH which uses the result of an earlier PUSH has the same context. For this reason, the entries (called buckets) in the WFST are keyed on the following information: (1) the state that is pushed for; (2) the lexical edge that is being considered;[24] (3) the list of SENDR registers; and (4)

---

[24] If the state and edge were the only keys necessary, the WFST would be equivalent to the chart as used by Kay (1964). Kaplan has extended the chart in GSP to include SENDRs and hold lists (1975). The LUNAR parser WFST (Woods 1973a) used all of these keys (except that the input string was used instead of the edge as the LUNAR parser did not use a chart).

---

[25] More collapsing of processing is possible but only with considerably greater complexity. Note that the sending of features is not permitted. It could be supported by making the list of sent features another key in the well-formed substring table.

the hold list.[25] Each bucket in the WFST contains two pieces of information: the open configurations (OPENCFs) which have pushed and are waiting for a constituent to be popped, and the well-formed substrings (WFSs) resulting from POPs.

A bucket is created by the first push arc whose configuration keys to it. When this occurs, a new configuration is set up to start processing at the lower level. Whenever another configuration keys to the same bucket, (i.e. pushes for the same constituent at the same place in the input with the same hold list and SENDR registers), it attaches itself to the list of OPENCFs in the bucket and creates POPped configurations for each WFS already stored there. Whenever a configuration POPs to a bucket (this will be discussed further later), it creates a well-formed substring (WFS), attaches it to the list of WFSs and creates configurations which start each of the OPENCFS using the found WFSs. In this way each configuration which is waiting for a POP gets started once and only once for each alternative POP. In a strictly depth-first situation, the configurations which arrive after the first one would not need to attach themselves because all possible alternatives at the lower level will have been exhausted. However, SUSPEND actions can result in later POPs which must know _all_ of the configurations to continue.

An OPENCF has the following pertinent information: the configuration state which is the label of (the code compiled from) the arc actions and termination action of the PUSH arc, a stack, a list of registers, a list of features, and a HOLD list. A WFS has: the structure being POPped together with its features, a node which indicates how much of the input was accepted during the PUSH, a list of lifted registers, a list of lifted features and a hold list. An OPENCF and a WFS are joined together to get a continuation configuration in the following manner. The state of the new configuration is the state specified by the OPENCF which continues the parse at the higher level. The node is taken from the WFS which indicates where in the input parsing it is to continue. The stack is taken from the OPENCF. The register and feature lists are the result of merging the lifted registers and features in the WFS with the registers and features from the OPENCF. The hold list is the result of removing from the HOLD list of the OPENCF, all of the elements which are not on the HOLD list of the WFS (i.e. all of the elements which were removed by VIR arcs in the lower level).

The well-formed substring table determines the structure of the stack. Without a WFST, the stack is a list of configurations the first of which is waiting to be continued as soon this level pops. With a WFST, the stack is a pointer to a bucket in the WFST which contains all of the configurations which are waiting for this level. Notice that a bucket in the WFST is a refinement of a "state set" in the Earley parsing algorithm (1970). In particular, it utilizes the same collapsing of the stack which allows parsing of left recursive grammars.

The accessing of information in registers at a higher level (to determine, for example, the top-level verb during the recognition of a relative clause) is not allowed using this scheme of a well-formed substring table because there may be multiple higher levels (OPENCFs), some of which haven´t been created yet. Equivalent effects in the grammar can be generated by using SENDR´s which will create different buckets for differing information in the sent registers.

It is useful to examine the differences which result in the ATN machine as a result of the well-formed substring table. The only difference in the object code is a change in where control is transferred after a PUSH arc. Without the WFST, control is passed directly to the first arc of the lower state (see the section on code for PUSH arcs). With the WFST, an earlier PUSH may have already started a configuration at the lower level so control is passed to the place which starts the next configuration. The major difference between the WFST and the non-WFST versions is in the runtime functions DOPUSH and DOPOP which actually maintain the WFST and perform the operations described above.

Alternative Control Strategies

Some of the most exciting developments in the study of syntactic processing have been in the area of alternative control strategies. Kaplan (1974) has done an excellent job of factoring out the scheduling aspects of the control component of a syntactic processor from its analytic aspects while Marcus (1975) is exploring general heuristics to guide the scheduling process. While the present implementation only produces programs which do depth-first search, the compilation process in no way requires this particular control strategy and alternative strategies are planned. One control strategy which looks very interesting has been named "burst" mode. To process a particular state, all of its arcs are applied and the ones which could be taken generate configurations. This list of configurations is then passed to a (possibly user provided) selection function which picks one or more to be continued while the rest are placed on the alternative list. This would provide the user with dynamic control over the selection of configurations and permit explorations of various strategies for semantically and pragmatically guided parsing.

Chapter 7

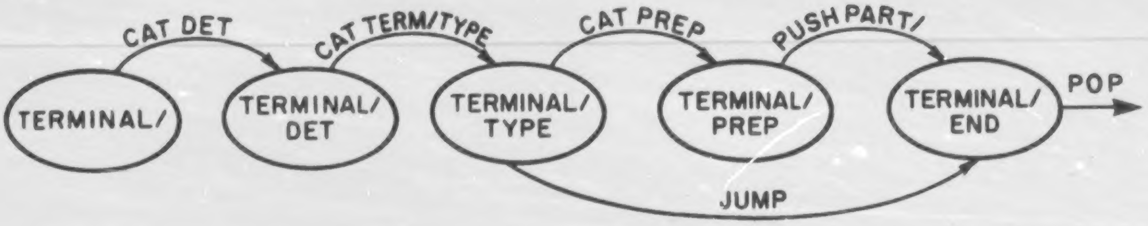
SEMANTIC ATN

For the reasons discussed in Chapter 5, the SOPHIE semantic grammar was re-written in the ATN formalism. We wish to stress here that the re-writing was a process of changing form only. The content of the grammar remained the same. Since a large part of the knowledge encoded by the grammar continues to be semantic in nature, we call the resulting grammar a "semantic ATN". Figure 7.1 presents the graphic ATN representation of a semantic grammar non-terminal. This is the same rule presented in Figure 4.1 which recognizes the phrases for specifying measurements in a circuit. The actions and structure building operations on the arcs (which are not shown in Figure 7.1) save the recognized constituents and construct the proper interpretation when sufficient information has been collected. Appendix G provides more examples of the semantic ATN used in SOPHIE.

Figure 7.2 presents a simple example of how the recognition of anaphoric deletions can be captured in ATN formalism. The network in Figure 7.2 encodes the straightforward way of expressing a terminal of a part in the circuit (e.g. the base of Q5, the anode of it, the collector.) By the state TERMINAL/TYPE, both the determiner and the terminal type (e.g. base, anode) have been found. The first arc leaving TERMINAL/TYPE accepts the preposition which begins the specification of the part. The second arc (JUMP arc) corresponds to hypothesizing that the specification of the part has been deleted (as in "The base is open"). The action on the arc builds a place holding form identifying the deletion and specifying (from information associated with the terminal type which was found) the classes of objects which can fill the deletion. The method for determining the referent of the deletion remains the same one described in Chapter 4.

Figure 7.2

An ATN which recognizes deletion

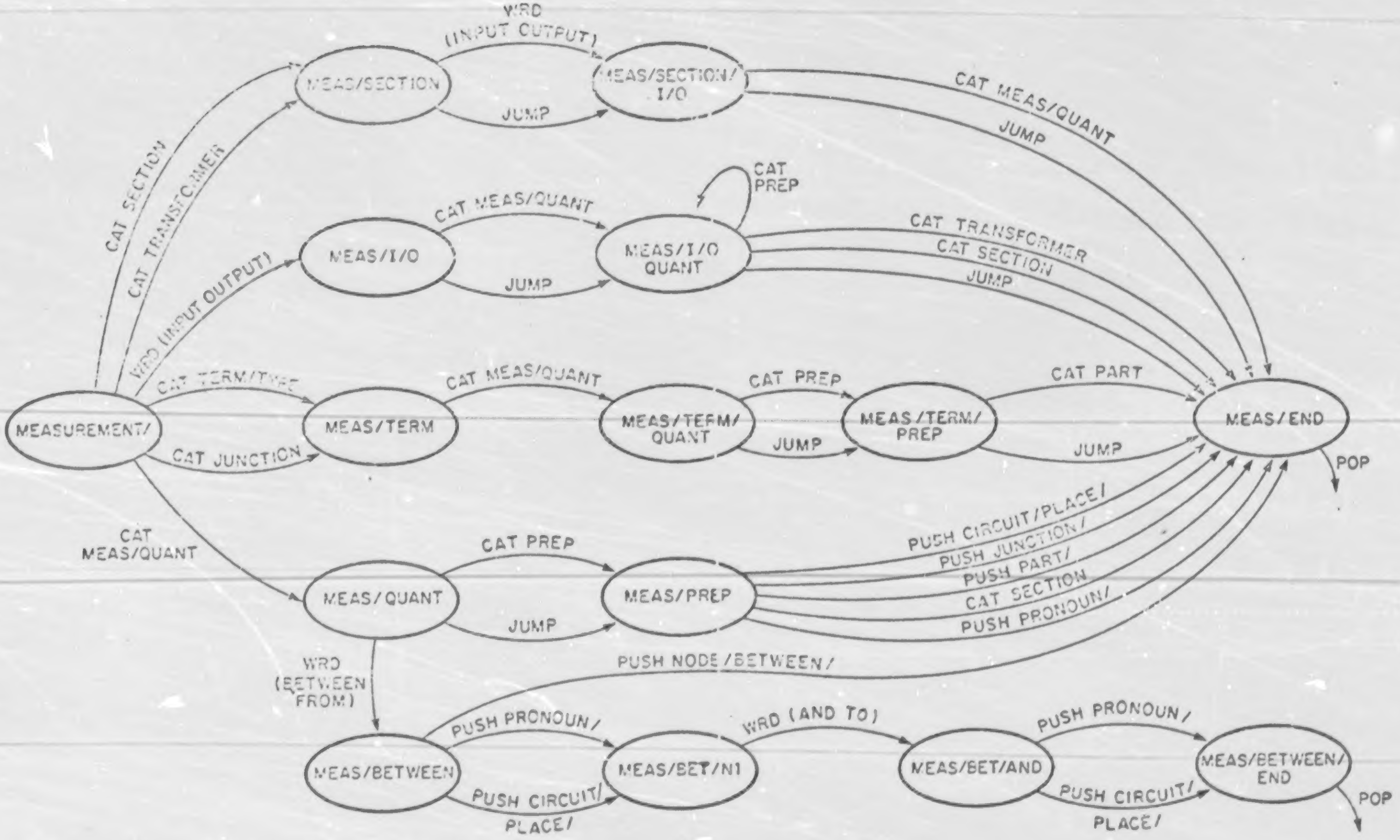


The SOPHIE semantic ATN is then compiled using the general ATN compiling system described in Chapter 6. The SOPHIE grammar provides the compiling system with a good contrast to the LUNAR grammar as it does not use many of the potential features. In addition, a bench mark, of sorts, was available from the LISP implementation of the grammar which could be used to determine the computational cost of using the ATN formalism.

There were two modifications made to the compiling system to improve its efficiency for the SOPHIE application. In the SOPHIE grammar, a large number of the arcs check for the occurrence of particular words. When there is more than one arc leaving a state, the ATN formalism requires that all of these arcs be tried, even if more than one of these is a WRD arc and an earlier WRD arc has succeeded. This is especially costly since the taking of an arc requires the creation of a configuration to try the remaining arcs. In those cases when it is known that none of the other arcs can succeed, this should be avoided. As a solution to this problem, the GROUP arc type was added. The GROUP arc allows a set of contiguous arcs to be designated as mutually exclusive. The form of the GROUP arc is (GROUP arc1 arc2 ... arcn). The arcs are tried one at a time until the conditions on one of the arcs are met. This arc is then taken and the remaining arcs in the GROUP are forgotten (not tried). If a PUSH arc is included in the GROUP, it will be taken if its test is true and the remaining arcs will not be tried even if the PUSHed for constituent is not found. For example, consider the following grammar state:

```
(S/1
    (GROUP (CAT A T (TO S/2))
```

FIGURE 7.1


MEASUREMENT/
MEAS/SECTION
MEAS/SECTION/I/O
MEAS/I/O
MEAS/I/O QUANT
MEAS/TERM
MEAS/TERM/QUANT
MEAS/TERM/PREP
MEAS/END
MEAS/QUANT
MEAS/PREP
MEAS/BETWEEN
MEAS/BET/N1
MEAS/BET/AND
MEAS/BETWEEN/END
WRD (INPUT OUTPUT)
JUMP
CAT MEAS/QUANT
CAT SECTION
CAT TRANSFORMER
WRD (INPUT OUTPUT)
CAT MEAS/QUANT
CAT PREP
CAT TRANSFORMER
CAT SECTION
CAT TERM/TYPE
CAT JUNCTION
CAT PREP
CAT PART
POP
CAT MEAS/QUANT
PUSH CIRCUIT/PLACE/
PUSH JUNCTION/
PUSH PART/
CAT SECTION
PUSH PRONOUN/
WRD (BETWEEN FROM)
PUSH NODE/BETWEEN/
PUSH PRONOUN/
WRD (AND TO)
PUSH CIRCUIT/ PLACE/

Example of a Semantic ATN

```
        (WRD X T (TO S/3))
        (CAT B T (TO S/4))))
```

At most one of the three arcs will be followed. Without GROUPing them together, it is possible that all three might be followed (i.e. if the word X had interpretations as both category A and category B).

The GROUP arc also provides an efficient means of encoding optional constituents. The normal method of allowing options in ATN is to provide an arc which accepts the optional constituent and a second arc which jumps to the next state without accepting anything. For example, if in state S/2 the word "very" is optional, the following two arcs would be created:

```
(S/2
    (WRD  VERY T (TO REST-OF-S/2))
    (JUMP REST-OF-S/2  T))
```

The inefficiency arises here when the word "very" does occur. The first arc is taken but an alternative configuration must be created (and possibly later explored) which will try the second arc. By embedding these arcs in a GROUP, the alternative will not be created (thus saving time and space) and hence won't have to be explored (possibly saving more time). A warning should be included here that the GROUP arc can cause sentences which might otherwise be accepted to be rejected. In our example, viewing "very" as a member of a category may be the only way out of the state REST-OF-S/2. In this respect the GROUP arc is a departure from the original ATN philosophy that arcs should be independent and for this we apologize. However, for some applications, the increased efficiency can be critical.

The other change to the compiling system for the semantic grammar application dealt with the preprocessing operations. The preprocessing facilities described in the last chapter included lexical analysis to extract word endings, a substitution mechanism to expand abbreviations, delete noise words, and canonicalize synonyms, dictionary retrieval routines and a compound word mechanism to collapse multi-word phrases. For the SOPHIE application we added the ability to use the INTERLISP spelling correction routines and the ability to derive word definitions from

SOPHIE's semantic net. The extraction of definitions from the semantic network for part names and node names reduces the size of the dictionary and simplifies the operations of changing circuits. In addition, a mechanism (called MULTIPLES) was developed which permits string substitution within the input. This is similar to the notion of compounding but differs in that a compound rule creates an alternative lexical item while multiple rule creates a different lexical item. After the application of a compound rule, there is an additional edge in the input chart while after a multiple rule, the effect is the same as if the user had typed in a different string.

Fuzziness

The one aspect of the LISP implementation which has not been incorporated into the ATN framework is fuzziness, the ability to ignore words in the input. While we have not worked out the details, the non-determinism provided by ATNs lends itself to an interesting approach. In a one-process (recursive descent) implementation, the rule which checks for a word must decide (with information passed down from higher rules) whether to try skipping a word or give up. The critical information which is not available when this decision has to be made is whether or not there is another parse which would use that word. In the ATN, it is possible to suspend a parse and come back to it after all other paths have been tried. Fuzziness could be implemented so that rather than skip a word and continue, it can skip a word and suspend, waiting for the other parses to fail or suspend. The end effect may well be that sentences are allowed to get fuzzier because there is not the danger of missing the correct parse.

Comparison of Results

The original motivation for changing to the ATN was its perspicuity. Appendices A and B which show the BNF/LISP version can be compared with Appendix G which shows the ATN version. We suspect that the reader finds neither of the two particularly readable, but then there is no reason to expect that this should be the case. As Winograd has pointed out (1973), simple grammars are perspicuous in almost any formalism; complex grammars are still complex in any formalism. In our own experience, we found the ATN formalism much easier to think in, to write in and to debug. The examples of redundant processing which were presented in chapter 5 were discovered while converting to ATN. For a gross comparison on conciseness, the ATN grammar requires 70% less characters to express than the LISP version.

The efficiency results were surprising. Table 7.1 gives comparison timings between the LISP version and the ATN compiled version. As can be seen, the LISP version is less than twice as fast. This was pleasantly counter-intuitive as we expected the LISP version to be much faster due to the amount of hand optimization which had been done while encoding the grammar rules. In presenting the comparison timing, it should be mentioned that there are three differences between the two systems which tended to favor the ATN version.[1] One difference was the lack of fuzziness in the ATN version. The LISP version spent time testing words other than the current word (looking ahead to see if it were possible to skip this word) which was not done in the ATN version. The second is the creation of categories for words during the preprocessing in the ATN version which reduced the amount of time spent accessing the semantic net and hence reduced the time required to perform a category membership test in the ATN system. The third was the simplification of the grammar and increase in the amount of bottom-up processing which could be done because of the ambiguity allowed in the input chart. In our estimation, the lack of fuzziness is the only difference which may have had a significant effect and this can be included explicitly in the ATN in places where it is critical (using TST arcs and suspend actions) without noticeable increase in processing time. In conclusion we are very pleased the results of the compiled semantic ATN and

---

[1] The exact extent to which each of these differences contributed is difficult to gather statistics on due to the block compiler which gains efficiency by hiding internal workings. The exact contribution of each could certainly be determined but was not deemed worth the effort.

Table 7.1

Comparison of ATN vs LISP Implementation

Times (in seconds) are "prepass" + "parsing"

1) What is the output voltage?

LISP - .024 + .018 = .042
ATN - .048 + .033 = .081

2) What is the voltage between there and the base of Q6?

LISP - .038 + .039 = .077
ATN - .090 + .046 = .136

3) Q5?

LISP - .010 + .046 = .056
ATN - .013 + .060 = .073

4) What is the output voltage when the voltage control is set to .5?

LISP - .045 + .038 = .083
ATN - .096 + .048 = .144

5) If Q6 has an open emitter and a shorted base collector junction what happens to the voltage between its base and the junction of the voltage limiting section and the voltage reference source?

LISP - .206 + .188 = .394
ATN - .259 + .090 = .349

feel that the ATN compiler makes the ATN formalism computationally efficient enough to be used in real systems.

Chapter 8

CONCLUDING DISCUSSION

When we began developing a natural language processor for an educational environment, we knew it had to be (1) fast, (2) habitable, and (3) self-teaching. The basic conclusion that has arisen from the work presented here is that it is possible to satisfy these constraints. The notion of semantic grammar (presented in Chapter 4) provides a paradigm for organizing the knowledge required in the understanding process which permits efficient parsing. In addition, semantic grammar aids the habitability by providing insights into a useful class of dialogue constructs and permits efficient handling of such phenomena as pronominalizations and ellipsis. The need for a better formalism for expressing semantic grammars led to the use of Augmented Transition Networks (presented in Chapter 6). This, in turn, led to the design and implementation of a general ATN compiler which drastically increased the speed of executing an ATN by translating it into an optimized object program. The increased efficiency makes practical the use of the ATN formalism for writing semantic grammars. The ability of the ATN-expressed semantic grammar to satisfy the above stated requirements is demonstrated in the natural language front-end for the SOPHIE system.

A point which needs to be stressed is that the SOPHIE system has been (and is being) used by uninitiated students in experiments to determine the pedagogical effectiveness of the SOPHIE environments. While much has been learned about the problems of using a natural language interface, these experiments were not "debugging" sessions for the natural language component. The natural language component has unquestionably reached a state at which it can be conveniently used to facilitate learning about electronics. In the remainder of the chapter, we will describe experiences of students using the natural language component and present research areas in which further work is necessary.

Impressions, Experiences and Observations

Prior to any exposure to SOPHIE, a group of four students were asked to write down all of the ways (they could think of) of requesting the voltage at a particular node. Although the intent of the experiment was to determine the range of paraphrases which students might be inclined to use before they were aware of the system's linguistic limitations, a more interesting result emerged. Each student wrote down one phrasing very quickly but had a difficult time thinking of a second, even though the initial phrasing by three of the students were in fact different! One student even gave up exclaiming "But there is only one way to ask that!" This same inability to perform linguistic paraphrase carried over to the actual interaction with SOPHIE via terminal. Whenever the system did not accept a query, there was a marked delay before the student tried again. Sometimes the student would abandon his line of questioning completely. At the same time, data collected over many sessions indicated that there was nothing like a standard (canonical) way to phrase a question. Table 8.1 provides some examples of the range of phrasings used by students to ask for the voltage at a node.

Table 8.1
Sample Student Inputs

The following are some of the input lines typed by students with the intent of discovering the voltage at a node in the circuit.

```
What is the voltage at node 1?
What is the voltage at the base of Q5?
How much voltage at N10?
And what is the voltage at N1?
N9?
V at the neg side of C6?
V11 is?
What is the voltage from the base of transistor Q5 to ground?
What V at N16?
Coll. of Q5?
Node 16 Voltage?
What is the voltage at pin 1?
Output?
```

As Table 8.1 shows, students are likely to conceive of their questions in many ways and to express each of these conceptions in any of several phrasings. Yet other experiences indicate that they lack the ability to easily convert to another conceptualization or phrasing. Since the non-acceptance of questions creates a major interruption in the student´s thought process, the acceptance of many different paraphrases is critical to maintaining flow in the student´s problem solving.

Another interesting phenomenon which occurred during sessions was the change in the linguistic behavior of the students as they used the system. Initially, queries were stated as complete English questions, generally stated in templates created by the students from the written examples of sessions which we had given them. If they needed to ask something which did not exactly fit one of their templates, they would try a minor variant. As they became more familiar with the mode of interaction, they began to use abbreviations, to leave out parts of their questions and, in general, to assume that the system was following their interaction. After five hours of experience with the system, almost all of one student´s queries contained abbreviations and one in six depended on the context established by previous statements.

RESEARCH AREAS IN SEMANTIC GRAMMAR

The SOPHIE semantic grammar system is designed for a particular context (trouble shooting) within a particular domain (electronics). It represents the compilation of those pieces of knowledge which are general (i.e. linguistic) together with specific domain dependent knowledge. In its present form, it is unclear which knowledge belongs to which area. The development of semantic grammars for other applications and extensions to the semantic grammar mechanism to include other understood linguistic phenomena will clarify this distinction.

While the work presented in this report has dealt mostly on one area of application, the notion of semantic grammar as a method of integrating knowledge into the parsing process has wider applicability. Two alternative applications of the technique have been completed. One deals with simple sentences in the domain of attribute blocks (Brown et al 1975). While the sublanguage accepted in the attribute blocks environment is very simple, it is noteworthy that within the semantic grammar paradigm a simple grammar was quickly developed which greatly improved the flexibility of the input language. The other completed application deals with questions about the editing system NLS (Grignetti et al 1975). In this application most questions dealt with editing commands and their arguments, and fit nicely into the case frame notion mentioned in Chapter 5. The case frame use of semantic grammar is being considered for (and may have its greatest impact on) command languages. Command languages are typically case centered around the command name which requires additional arguments (its cases). The combination of the semantic classification provided by the semantic grammar and the representation of case rules permitted by ATNs should go a long way towards reducing the rigidity of complex command languages such as those required for message processing systems. The combination should also be a good representation for natural language systems in domains where is it possible to develop a strong underlying conceptual space, such as management information systems (Malhotra 1975).

The extension of the semantic grammar to incorporate existing linguistic processing techniques is another potentially fruitful research area. One of the ways semantic grammar gains efficiency is to separate processing of syntactically similar sentences on semantic grounds when useful to do so. However, this prevents the uniform incorporation of, for example, Woods´ (1973b) solution to the problems of relative clause modification, quantifiers and conjunction. One means of integrating these techniques would be to develop an intermediate target language which maintains the advantages of the semantic grammar approach while allowing uniform solutions to other problems. It may even be possible to adopt Woods´ query language, allowing the semantic grammar to dictate the functions within the "propositions" and "commands". An alternative attack would be to use a "syntactic" processing phase, incorporating the desired techniques which canonicalizes the input before it is processed by the semantic grammar. In this method, the semantic grammar would be viewed as an interpretation phase of the understanding process, but which works on a much less structured syntactic parse than, for example, the LUNAR system.

FEEDBACK - When the Grammar Fails

A much neglected research area in natural language systems is the problem of providing feedback to unacceptable inputs (i.e. what to do when the system doesn´t understand an input). While it may appear that in a completely habitable system all inputs would be understood, no system has ever attained this goal and none will in the foreseeable future. To be natural to a naive user, an intelligent system should act intelligent when it fails too. The first step towards having a system fail intelligently is the identification of possible areas of error. In student´s use of the SOPHIE system, we have found the following types of error common:

(1) Spelling errors and mis-typings - "Shortt the CE og Q3 and opwn its base"; "What isthe VBE Q5?"

(2) Inadvertent omissions - "What is the BE of Q5?" (The user left out the quantity to measure. Note that in other contexts this is a well formed question.)

(3) Slight misconceptions which are predictable - "What is the output of transistor Q3?" (The output of a transistor is not defined); "What is the current thru node 1?" (Nodes are places where voltage is measured and may have numerous wires associated with them); "What is R9?" (R9 is a resistor); "Is Q5 conducting?" (The laboratory section of SOPHIE gives information which is directly available from a real lab such as currents and voltages.)

(4) Gross misconceptions whose underlying meaning is well beyond designed system capabilities - "Make the output voltage 30 volts"; "Turn on the power supply and tell me how the unit functions"; "What time is it?".

The best technique for dealing with each type of error is an open problem. In the remainder of this section, we will describe the solutions used in the SOPHIE system and present their shortcomings.

The use of spelling correction algorithm (borrowed from INTERLISP) has proven to be a satisfactory solution to errors of type 1. During one student´s session, spelling correction was required on (and resulted in proper understanding of) 10% of the questions. The major failings of the INTERLISP algorithm are the restriction on the size of the target set of correct words (time increases linearly with the number of words) and its failure to correct run-on words. (The time required to determine if a word may be two (possibly misspelled) words run together increases very quickly with the length of the word and the number of possibly correct words. With no context to restrict the possible list of words, the computation involved is prohibitive.) A potential solution to both shortcomings would be to use the context of the parser when it reaches the unknown word to reduce the possibilities. Because of the nature of the grammar, this would allow semantic context as well as syntactic context to be used.

Of course, the use of any spelling correction procedure has some dangers. A word which is correctly spelled but which the system doesn't know may get spelling corrected to a word the system does know. For example if the system doesn't know the word "top" but does know "stop", a user's command to "top everything" can be disastrously misunderstood. For this reason, words like "stop" are not spelling corrected.

Our solution to predictable misconceptions (type 3 errors) is to recognize them and give error messages which are directed at correcting the misconception. We are currently using two different methods of recognition. One is to loosen up the grammar so that it accepts plausible but meaningless sentences. This technique provides the procedural specialists called by the plausible parse enough context to make relevant comments. For example, the concept of current through a node is accepted by the grammar even though it is meaningless. The specialist which performs measurements must then check its arguments and provide feedback if necessary:

```
>> WHAT IS THE CURRENT THRU NODE 4?
The current thru a node is not meaningful since by Kirchoff's law
the sum of the currents thru any node is zero. Currents can be
measured thru parts (e.g. CURRENT THRU C5) or terminals
(e.g. CURRENT THRU THE COLLECTOR OF Q2).
```

Notice that the response to the question presents some examples of how to measure the currents along wires which lead into the mentioned node. Examples of questions which will be accepted and are relevant to the student's needs are among the best possible feedback.

The second method of recognizing common misconceptions is to "key" feedback off single words or groups of words. In the following examples, the "keys" are "or" and "turned on". Notice that the response presents a general characterization of the violated limitations as well as suggestions for alternative lines of attack.

>> COULD Q1 OR Q2 BE SHORTED?

I can only handle one question, hypothesis, etc. at a time. The fact that you say 'OR' indicates that you may be trying to express two concepts in the same sentence. Maybe you can break your statement in two or more simple ones.

>> IS THE CURRENT LIMITING TRANSISTOR TURNED ON?

The laboratory section of SOPHIE is designed to provide the same elementary measurements that would be available in a real lab. If you want to determine the state of a transistor, measure the pertinent currents and voltages.

These methods of handling type 3 errors has proven to be very helpful. However, they have the major drawback that all of the misconceptions must be predicted and programmed for in advance. This limitation makes them inapplicable to novel situations.

The most severe problems a user has stem from type 2 (omissions) and type 4 errors (major misconceptions). (Type 3 errors which haven't been predicted are considered type 4 errors.) After a simple omission, the user may not see that he has left anything out and may conclude that the system doesn't know that concept or phrasing of that concept. For example when the user types "What is the BE of Q5" instead of "What is the VBE of Q5?", he may decide that it is unacceptable because the system doesn't allow "VBE" as an abbreviation of "base emitter voltage". For type 4 errors, the user may waste a lot of time and energy attempting several rephrasings of his query, none of which can be understood because the system doesn't know the concept the user is trying to express. For example, no matter how it is phrased the system won't understand "Make the output voltage 30 volts" because measurements aren't things which can be directly changed, only controls and specifications of parts can be changed.

The feedback necessary to correct both of these classes of errors must identify any concepts in the statement which are understood and suggest the range of things which can be done to/with these concepts. For type 2 errors, this will help the user see his omission. For type 4 errors, it may suggest alternative conceptualizations which allow the user to get at the same information (for example, to change the output voltage indirectly by changing one of the controls) or at least provide him with enough information to decide when to give up.

The notion of semantic grammar may be useful in developing a general solution along the following lines: A bottom-up or island parsing scheme could be used to identify well-formed constituents.[1] Since the grammar is semantically based the constituents which are found represent "islands" of meaningful phrases. The ATN representation of the semantic grammar can then be inspected to discover possible ways of combining these islands. If a good match is found, the grammar can be used to generate a response which indicates what other semantic parts are required for that rule. Even if no good matches are found, a positive statement may be made which explains the set of possible ways the recognized structures could be understood. Much more work is required in the area of unacceptable inputs before natural language systems will feel really natural to naive users.

FUTURE RESEARCH AREAS IN ATN COMPILATION

There are several directions in which the ATN compilation system could be extended. One which was mentioned in Chapter 6 is the implementation of alternative control strategies. One example is the burst mode strategy. The burst mode strategy creates all possible configurations which could follow the current one and provides for user selection of which alternative to process next. This allows the user to discover the selection function which best serves his usage. The implementation of burst mode within the compiling system framework should not be very difficult. The code from a state would create a list of the possible configurations which result from that state. That is, for all arcs in a state, if the arc condition is satisfied, a new configuration is made by copying the present one, and the code compiled from the arc actions is executed. The arc action code changes the new configuration and the code from the arc termination action adds the changed configuration to the possible-next configurations list. After all of the arcs have been tested, control is passed to a (possibly state dependent) selection function which chooses one or more configurations to continue and places the remainder on the list of alternatives. When compared to depth-first, this strategy has the disadvantage that configurations cannot be cannabilized. However, it has the advantage that configurations are only created for possibly successful paths. (In depth-first, a configuration is needed to remember the alternative arcs from a state even if none of them will succeed.) Burst mode also allows dynamic selection of search strategy.

---

[1] William Woods and Geoff Brown are presently refining such a bottom-up parsing technique for ATN grammars for use in the BBN Speech project (Woods 1976).

At present, the ATN compiler produces only LISP object code, however, the ATN object code does not place heavy restrictions on the choice of object language. The necessary constructs are: conditional statements, a function calling mechanism and list processing routines. The generation of ALGOL, BCPL or even machine language code would present no major technical difficulties. A large effort would have to be expended, however, implementing the necessary runtime environment (lexical routines, configuration management routines, arc actions, etc.) for the resulting ATN machine. For production environments or situations where more speed was essential, this effort may prove worthwhile. An advantage of this method of implementing, say, a command scanner is that the development and debugging can be done in the INTERLISP programming environment at very little cost to the efficiency of the final product. This technique could also be used to develop parsing programs for mini-computers or "intelligent" terminals.

CONCLUSIONS

In the course of this report, we have described the evolution of a natural language front-end from keyword beginnings to a system capable of using complex linguistic knowledge. The guiding strand has been the utilization of semantic information to produce efficient natural language processors. During the evolution of the system, there are several highlights which represent noteworthy points in the spectrum of useful natural language systems. Toward the keyword end of the scale, the procedural encoding technique with fuzziness (Chapter 4 and Appendix B) allows simple natural language input to be accepted without introducing the complexity of a new formalism. Encoding the rules as procedures allows flexible control of the fuzziness and the semantic nature of the rules provides the correct places to take advantage of the flexibility. As the language covered by the system become more complex, the additional burden of a grammar formalism will more than pay for itself in terms of ease of development and reduction in complexity. The ATN compiling system allows the consideration of the ATN formalism by reducing its runtime cost making it comparable to a direct procedural encoding. The natural language front end now used by SOPHIE is constructed by compiling a semantic ATN. As the linguistic complexity of the language accepted by the system increases, the need for more syntactic knowledge in the grammar becomes greater. Unfortunately, this often works at cross purposes with the semantic character of the grammar. It would be nice to have a general grammar for English syntax which could be used to preprocess sentences, however one is not forthcoming. A general solution to the problem of incorporating semantics with the current state of incomplete knowledge of syntax remains an open research problem. In the foreseeable future, any system will have to be an engineering trade-off between complexity and generality on one hand and efficiency and habitability on the other. We have presented several techniques which are viable bargains in this trade-off.

References

Bates, M. "Syntactic Analysis in a Speech Understanding System." BBN Report No. 3116, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1975.

Bates, M. and W. Woods, "The Syntactic Component." in "Speech Understanding Research at BBN." BBN Report No. 2976. Bolt Beranek and Newman Inc., Cambridge, Massachusetts, December 1974.

Bobrow, D.G. "A Note on Hash Linking." *Communications of the ACM*. 18(1975), 413-415.

Bobrow, D.G. and A. Collins, Eds. *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press, 1975.

Bobrow, R.J. and J.S. Brown, "Systematic Understanding: Synthesis, Analysis, and Contingent Knowledge in Specialized Understanding Systems." *Representation and Understanding: Studies in Cognitive Science*. Eds. D. Bobrow and A. Collins. New York: Academic Press, 1975.

Brown, J.S. and R.R. Burton, "Multiple Representations of Knowledge for Tutorial Reasoning." *Representation and Understanding: Studies in Cognitive Science*. Eds. D. Bobrow and A. Collins. New York: Academic Press, 1975.

Brown, J.S., R.R. Burton, and A.G. Bell, "SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting (An Example of AI in CAI)." BBN Report No. 2790, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, March 1974.

Brown, J.S., R.R. Burton, and A.G. Bell, "SOPHIE: A Step Towards a Reactive Learning Environment." *International Journal of Man Machine Studies*. 7(1975), 675-696.

Brown, J.S., R.R. Burton, M. Miller, J. DeKleer, S. Purcell, C. Hausmann and R.J. Bobrow, "Steps Toward a Theoretical Foundation for Complex Knowledge-Based CAI." Final Report, Bolt, Beranek and Newman Inc., Cambridge, Massachusetts, 1975.

Brown, J.S., R.R. Burton, and F. Zdybel, "A Model-driven Question Answering System for Mixed-initiative Computer Assisted Instruction." *IEEE Transactions on Systems, Man and Cybernetics*. 3(1973).

Brown, J.S., R. Rubinstein, and R.R. Burton, "Reactive Learning Environment for Computer Assisted Electronics Instruction." BBN Report No. 3314, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, October 1976.

Bruce, B.C. "Case Systems for Natural Language." *Artificial Intelligence*. December 1975. 327-360.

Charniak, E. "Toward a Model of Children's Story Comprehension." MIT--TR-266, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1972.

Chomsky, N. *Syntactic Structures*. The Hague: Mouton and Co., 1957.

Chomsky, N. *Aspects of the Theory of Syntax*. Cambridge, Massachusetts: The MIT Press, 1965.

Codd, E.F. "Seven Steps to Rendezvous With the Casual User." Proceedings of the IFIP TC-2 Working Conference on Data Base Management Systems. Amsterdam, 1974.

Colby, K.M., "Simulation of Belief Systems." *Computer Models of Thought and Language*. Eds. R.C. Schank and K.M. Colby. San Francisco: W.H. Freeman and Company, 1973.

Colby, K.M., R.C. Parkinson, and B. Fraught, "Pattern Matching Rules for the Recognition of Natural Language Dialogue Expressions." *American Journal of Computational Linguistics*. Microfiche 5, 1974.

Coles, L.S. "Syntax Directed Interpretation of Natural Language." *Representation and Meaning: Experiments With Information Processing Systems*. Eds. H.A. Simon and L. Siklossy. Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

Earley, J. "An Efficient Context-Free Parsing Algorithm." *Communications of the ACM*. 13(1970), 94-102.

Goldberg, A. "Computer-assisted Instruction: The Application of Theorem Proving to Adaptive Response Analysis." Technical Report No. 203, Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Goldstein, I.P. "Understanding Simple Picture Programs" MIT-AI-TR-294, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1974.

Grignetti, M.C., L. Gould, C.L. Hausmann, A.G. Bell, G. Harris and J. Passafiume. "Mixed-Initiative Tutorial System to Aid Users of the On-Line System (NLS)." BBN Report No. 2969, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, November 1974.

Grignetti, M.C., C. Hausmann, and L. Gould, "An ´Intelligent´ On-line Assistant and Tutor - NLS-SCHOLAR." *National Computer Conference*. 1975. 775-781.

Heidorn, G.E. "Natural Language Inputs to a Simulation Programming System." Technical Report NPS-55HD72101A, Naval Postgraduate School, Monterey, California. 1972.

Heidorn, G.E. "English as a Very High Level Language for Programming." *Proceedings of the Symposium on Very High Level Languages*. SIGPLAN Notices 9, 1974, 91-100.

Heidorn, G.E. "Augmented Phrase Structure Grammars." *Proceedings of a Workshop on Theoretical Issues in Natural Language Processing*. Eds. R. Schank and B.L. Nash-Webber. 1975. 1-5.

Irons, E.T. " A Syntax Directed Compiler for ALGOL 60." *Communications of the ACM*. 4(1961), 51-55.

Kaplan, R.M. "A General Syntactic Processor." *Natural Language Processing*. Ed. Randall Rustin. New York: Algorithmics Press, 1973.

Kaplan, R.M. "Transient Processing Load in Relative Clauses." Doctoral Dissertation, Psychology Department, Harvard University, 1974.

Kaplan, R.M. Personal communication. 1975.

Kay, M. "Experiments With a Powerful Parser." RM-5452-PR. The Rand Corporation, Santa Monica, California. 1967.

Kay, M. Personal communication. 1973.

Klovstad, J.W. CASPERS, Computer Automated Speech Perception System, Doctoral Dissertation, M.I.T., 1977.

Malhotra, A. "Design Criteria for a Knowledge-Based English Language System for Management: An Experimental Analysis" Doctoral Dissertation, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts, February, 1975.

Marcus, M. "Diagnosis as a Notion of Grammar." *Proceedings of a Workshop on Theoretical Issues in Natural Language Processing*. Eds. R. Schank and B.L. Nash-Webber. 1975. 6-10.

Miller, R.B. "Response Time in Man-computer Conversational Transactions." *AFIPS Conference Proceedings*. Fall Joint Computer Conference. Washington: Thompson Book Company, 1968, 267-278.

Quillian, M.R. "The Teachable Language Comprehender: a simulation program and theory of language." *Communications of the ACM*. 12(1969), 459-476.

Rustin, R. Ed. *Natural Language Processing*. New York: Algorithmics Press, 1973.

Schank, R.C. and K.M. Colby, Eds. *Computer Models of Thought and Language*. San Francisco: W.H. Freeman and Company, 1973.

Schank, R.C., M.N. Goldman, C.J. Reiger, and C.K. Riesbeck, "Inference and Paraphrase by Computer." *Journal of the ACM*. 3(1975), 309-328..

Shapiro, S.C. and S.C. Kwasny, "Interactive Consulting via Natural Language." *Communications of the ACM*. 18(1975), 459-462.

Simmons, R.F. "Natural Language Question-Answering Systems: 1969." *Communications of the ACM*. 13(1970), 15-30.

Simmons, R.F. "Semantic Networks: Their Computation and Use for Understanding English." in *Computer Models of Thought and Language*. Eds. R.C. Schank and K.M. Colby. San Francisco: W.H. Freeman and Company. 1973.

Simon, H.A. and L. Siklossy, Eds. *Representation and Meaning: Experiments with Information Processing Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

Smith, N.W. "A Question-answering System for Elementary Mathematics." Technical Report No. 227, Institute for Mathematical Studies in the Social Sciences, Stanford University. 1974.

Smith, R.L., N.W. Smith, and F.L. Rawson, "CONSTRUCT: In Search of a Theory of Meaning." Conference of the Association for Computational Linguistics. Amherst, Massachusetts. 1974.

Teitelman, W. "Towards a Programming Laboratory." *International Joint Conference on Artificial Intelligence*. Ed. D. Walker. May 1969.

Teitelman, W. "Automated Programming - The Programmer's Assistant." *Proceedings of the Fall Joint Computer Conference*. December 1972.

Teitelman, W. "CLISP-Conversational LISP." *Third International Joint Conference on Artificial Intelligence*. August 1973.

Teitelman, W. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, California, 1974.

Watt, W.C. "Habitability." American Documentation. 19(1968), 338-351.

Weizenbaum, J. "ELIZA -- A Computer Program for the Study of Natural Language Communication Between Man and Machine." Communications of the ACM. 9(1966), 36-43.

Weizenbaum, J. "Contextual Understanding by Computers." Communications of the ACM. 10(1967), 474-480.

Wilks, Y. "The Stanford Machine Translation Project." Natural Language Processing. Ed. Randall Rustin. New York: Algorithmics Press, 1973a.

Wilks, Y. "An Artificial Intelligence Approach to Machine Translation." Computer Models of Thought and Language. Eds. R.C. Schank and K.M. Colby. San Francisco: W.H. Freeman and Company, 1973b.

Wilks, Y. "Natural Language Systems Within the AI Paradigm: A Survey and Some Comparisons." Stanford Artificial Intelligence Laboratory Memo AIM-237, Computer Science Department, Stanford. 1974.

Winograd, T. Understanding Natural Language. New York: Academic Press, 1972.

Woods, W.A. "Semantics for a Question-Answering System." Doctoral Dissertation, Harvard University, Cambridge, Massachusetts, 1967.

Woods, W.A. "Procedural Semantics for a Question-Answering Machine." AFIPS Conference Proceedings. 33(1968).

Woods, W.A. "Augmented Transition Networks for Natural Language Analysis." Harvard Computation Laboratory Report No. CS-1, Harvard University, Cambridge, Massachusetts. 1969.

Woods, W.A. "Transition Network Grammars for Natural Language Analysis." Communications of the ACM. 13(1970), 591-606.

Woods, W.A. "An Experimental Parsing System for Transition Network Grammars." Natural Language Processing. Ed. Randall Rustin. New York: Algorithmics Press, 1973a..

Woods, W.A. "Progress in Natural Language Understanding - An Application to Lunar Geology." National Computer Conference. 1973b. 441-450.

Woods, W.A. "What's In a Link: Foundations for Semantic Networks." in Representation and Understanding: Studies in Cognitive Science. Eds. D. Bobrow and A. Collins. New York: Academic Press, 1975.

Woods, W.A., R.M. Kaplan, and B. Nash-Webber, "The Lunar Sciences Natural Language Information System: Final Report." BBN Report 2378, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, 1972.

Woods, W. Personal communication. 1976.

Appendix A

BNF Description of Part of the SOPHIE Semantic Grammar

This appendix gives a BNF-like description of part of the language accepted by SOPHIE. Included are all of the rules necessary to parse a "measurement". Examples of "measurements" are "voltage at N1", "base emitter current of Q5", and "output voltage". The grammar is implemented as LISP functions and an example is listed in Appendix B.

In the description, alternatives on the right-hand side are separated by ! or are listed on separate lines. Brackets [] enclose optional elements. An asterisk * is used to mark notes about a particular rule. Non-terminals are designated by names enclosed in angle brackets <>.

The Grammar

```
<circuit/place>:= <terminal> ! <node>

<diode/spec> := <diode> ! <zener/diode>
                <section> diode ! <section> zener/diode

<junction> := <junction/type> [of] <transistor/spec>
              <transistor/term/type> and <transistor/term/type> [of]
                    [<transistor/spec>]
              <transistor/term/type> to <transistor/term/type> [of]
                    [<transistor/spec>]

<junction/type> := eb ! be ! ec ! ce ! cb ! bc

<meas/quant> := voltage ! current ! resistance* ! power
                *means measured resistance

<measurement> := <section>[output*][<meas/quant>]
                 output* <meas/quant> [of] <section>
                 output* [<meas/quant>] [of <transformer>]
                 <transformer> <meas/quant>
                 <meas/quant> between** <circuit/place> and*
                   <circuit/place>
                 <meas/quant> of*** <part/spec>
                 <meas/quant> between output terminals
                 <meas/quant> of <junction>
                 <meas/quant> of <circuit/place>
                 <meas/quant> from <junction>
                 <meas/quant> of <section>
                 <meas/quant> of <pronoun>
                 <junction/type> <meas/quant> [of <transistor/spec>]
                 <transistor/term/type> <meas/quant> of
                   [<transistor/spec>]
                 *input also
                 **from-to also works
                 ***at, thru, in, into, across and through also work

<node> := junction of <part/spec> and <part/spec>
          node between <section> and <section>
          [point] between <part/spec> and <part/spec>
          <node/name> ! [node] <node/number>
          <pronoun>

<num/spec> := "any positive number" [k] ! one

<part/spec> := <part/name> ! <load/spec> ! <section> <part/type>
               <pronoun>
```

```
<pot/spec> := cc ! vc ! cct

<pronoun>:= it ! [that] "type"

<terminal> := output [terminal] ! <transistor/term> ! center/tap
              positive terminal [<part/spec>] ! positive one
              negative terminal [<part/spec>] ! negative one
              anode [<diode/spec>] ! cathode [<diode/spec>]
              wiper [<pot/spec>]

<transistor/spec> := <transistor> ! <section> transistor ! <pronoun>

<transistor/term> := <transistor/term/type> [<transistor/spec>]

<transistor/term/type> := base ! collector ! emitter
```

<transistor>, <capacitor>, <diode>, <resistor>, <transformer> and <zener/diode> all check the semantic network and parse correct part names, e.g. r9, q6.

<section> uses the semantic network to determine if a word is a section of the unit, e.g. current/limiter.

<part/name> uses the semantic network to see if a word is the name of a part e.g. r6, c4, t2.

<node/name> checks semantic network for node names.

Appendix B

A LISP Rule from the Semantic Grammar

This appendix describes the method of encoding the grammar as LISP procedures. The ways of expressing a non-terminal are embodied in a grammar function. Each grammar function takes at least two arguments; STR, the list of words to be recognized, and N, the degree of fuzziness allowed. The grammar function, in effect, must determine whether the beginning of the string STR contains an occurrence of the corresponding non-terminal. There are generally two types of checks that a grammar function performs. One is a check for the occurrence of a word or words which satisfies certain predicates. This checking is done with two functions -- CHECKLST and CHECKSTR. CHECKLST looks for a word in the string matching any of a list of words. CHECKSTR looks for a word in the string satisfying an arbitrary predicate. It is through these functions that the parser implements its fuzziness. For example, if CHECKSTR is called with the string "resistor R9" and a predicate which determines if a word is the name of a part (e.g. "R9"), CHECKSTR will succeed by skipping the word "resistor", which in this phrase, is a noise word.

The other usual type of operation performed by the grammar functions is to check for the occurrence of other non-terminals. This is done by calling the proper function (grammar rule) and passing it the correct position in the input string.

If a grammar rule is successful, the function passes back two pieces of information. First, it returns some indication of how much of the input string is accepted (i.e. where it stopped). The convention adopted is that the grammar rule returns as its value a pointer to the last word in the string accepted by the rule. Second, the function passes back a structural description of the phrase that was parsed. This structure is passed back in the free variable RESULT (analogous to an ATN's "*" upon return from a PUSH.

Listed below is the grammar rule for the concept of a junction of a transistor. This rule accepts phrases such as "base emitter junction of Q5", "BE of the current limiting transistor", or "collector emitter junction".

```
(<JUNCTION>
 [LAMBDA (STR N)
  (PROG (TS1 R1)
      (RETURN
      (AND

(* COMMENT A)

        [OR (AND (SETQ TS1 (<JUNCTION/TYPE> STR N))
                 (SETQ R1 RESULT))
            (AND (SETQ TS1 (<TRANSISTOR/TERM/TYPE> STR N))
                 (SETQ R1 RESULT)
                 [SETQ TS1
                   (<TRANSISTOR/TERM/TYPE>
                     (CDR (CHECKLST (CDR TS1)
                                    (QUOTE (AND TO]
                 (SETQ R1 (JUNCTION-OF-TERMS R1 RESULT]

(* COMMENT B)

        (COND
          ([SETQ STR (<TRANSISTOR/SPEC>
                (CDR (GOBBLE (GOBBLE TS1 (QUOTE (JUNCTION)))
                             (QUOTE (OF))
                              1]
            (SETQ RESULT (LIST R1 RESULT))
            STR)
```

```
([SETQ RESULT (LIST R1 (LIST (QUOTE PREF)
                             (QUOTE (TRANSISTOR]
  TS1]))
```

COMMENT A:

The first thing that is looked for is either a <junction/type> (BE, emitter collector, etc.) or two <transistor/terminal/type>s (base, emitter or collector) separated by the words "and" or "to". If two terminals are found, the function JUNCTION-OF-TERMS is called to determine the proper junction. In either case, the place where the successful subsidiary rule left off is saved in TS1 and the meaning of the accepted phrase is saved in R1.

COMMENT B:

The next thing needed for a junction is a transistor <TRANSISTOR/SPEC>. <TRANSISTOR/SPEC> looks for an occurrence of a transistor, e.g. "Q5" or "current limiting transistor". GOBBLE is a function for skipping relational words when they are not used to restrict the remaining part of the phrase. If a transistor is not found, a deletion is hypothesized and a call to PREF is constructed. If the transistor has been pronominalized as in "the base emitter of it", <TRANSISTOR/SPEC> would recognize "it". In either case the semantics of the recognized phrase (something like (EB Q5)) is put into RESULT and a pointer to the last recognized word is returned as the value of <JUNCTION>.

There are approximately 80 grammar rules in SOPHIE´s grammar.

Appendix C

Sample Parses and Parse Times for the LISP Implementation

This appendix presents some examples of sentences handled by the natural language processor together with their parse times. Under each statement, the semantic interpretation returned by the parser is given. The semantic interpretation is a function call which when evaluated performs the processing required by the statement. Parse times are given in milliseconds.

```
Insert a fault.
(INSERTFAULT NIL)
85 ms

What is the output voltage?
(MEASURE VOLTAGE NIL OUTPUT)
40 ms

What is the voltage between the current limiting transistor
and the constant current source?
(MEASURE VOLTAGE (NODE/BETWEEN
                    (FINDPART CURRENT/LIMITER TRANSISTOR)
                     CURRENT/SOURCE))
335 ms

What is the voltage between there and the base of Q6?
(MEASURE VOLTAGE (PREF (NODE TERMINAL)) (BASE Q6))
80 ms

Q5?
(REFERENCE ((TRANSISTOR) Q5))
60 ms

Could the problem be that Q5 is bad?
(TESTFAULT Q5 BAD)
100 ms

Could it be shorted?
(TESTFAULT (PREF (PART JUNCTION TERMINAL)) SHORT)
75 ms

If R8 were 30k what would the output voltage be?
(IFTHEN (R8 30000.0 VALUE)
        (MEASURE VOLTAGE NIL OUTPUT))
220 ms

If C2 were leaky what would the voltage across it be?
(IFTHEN (C2 LEAKY)
        (MEASURE VOLTAGE (PREF (PART JUNCTION)))
120 ms

What is the output voltage when the voltage control is set to .5?
(RESETCONTROL (STQ VC .5)
              (MEASURE VOLTAGE NIL OUTPUT))
85 ms

What is it with it set at .6?
(RESETCONTROL (STQ (PREF (POT LOAD SWITCH)) .6)
              (REFERENCE NIL))
110 ms

If it is set to .9?
(RESETCONTROL (STQ (PREF (POT LOAD SWITCH)) .9)
              (REFERENCE NIL))
 135 ms
```

what is the current thru the cc when the vc is set to 1.0?

```
(RESETCONTROL (STQ VC 1.0)
              (MEASURE CURRENT CC))
```

190 ms

If Q6 has an open emitter and a shorted base collector junction, what happens to the voltage between its base and the junction of the voltage limiting section and the voltage reference source?

```
(IFTHEN
      (MULT ((EMITTER Q6) OPEN)
            ((BC (PREF (TRANSISTOR))) SHORT))
      (MEASURE VOLTAGE
               (BASE (PREF (TRANSISTOR)))
               (NODE/BETWEEN VOLTAGE/LIMITER REFERENCE/VOLTAGE)))
```

400 ms.

Appendix D

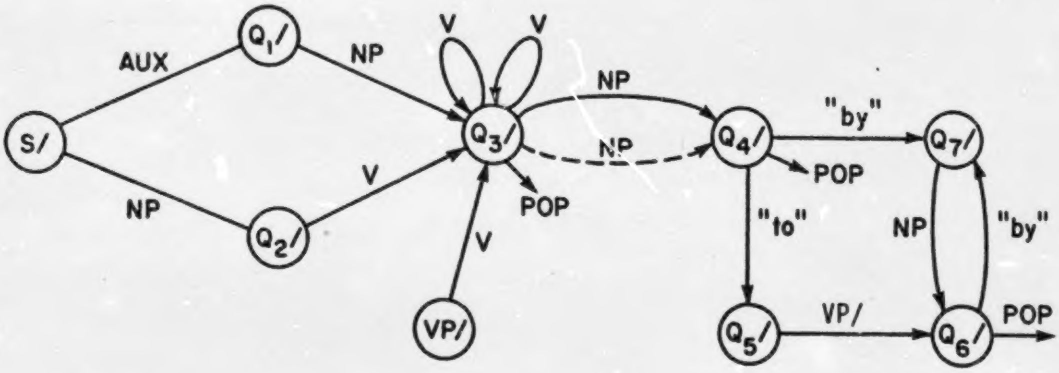
Examples of ATN Compilation

This appendix presents a simple augmented transition network grammar along with two different programs compiled from it and a trace of the first program parsing a sentence. The ATN grammar was taken from (Woods 1970). Both compiled versions of the grammar assume a depth-first search strategy and use configurations which include the state, node, stack, registers, features and hold list.

The first program does not support lexical ambiguity (neither that caused by compound rules nor that caused by multiple interpretations under the same category). In addition, it neither keeps a well-formed substring table, tests for input before pushing nor returns features with popped constituents. The second program, on the other hand, has all of these capabilities. The listing of the second program also includes tracing functions the compiler includes in the program to allow the user to follow its operation. Both programs are given in CLISP (Teitelman 1974).

The final section of the appendix contains a trace of the first program (using a version which did include tracing functions) discovering all possible parses of the sentence "John was believed to have been shot by Fred". Shown in the trace are all of the arc transitions taken by the parser together with all register setting operations. (The reader may compare this with the analysis of this sentence given in (Woods 1970).)

The grammar


S/
AUX
NP
Q1/
NP
Q2/
V
V
V
Q3/
POP
V
VP/
NP
NP
Q4/
POP
"by"
"to"
Q7/
NP
"by"
Q5/
VP/
Q6/
POP

```
(S/
   (CAT AUX T
        (SETR V *)
        (SETR TNS (LIST (GETF * TENSE)))
        (SETRQ TYPE Q)
        (TO Q1/))
   (PUSH NP/ T
        (SETR SUBJ *)
        (SETRQ TYPE DCL)
        (TO Q2/)))
(Q1/
   (PUSH NP/ T
        (SETR SUBJ *)
        (TO Q3/)))
(Q2/
   (CAT V T
        (SETR V *)
        (SETR TNS (LIST (GETF * TENSE)))
        (TO Q3/)))
(Q3/
   (CAT V (AND (GETF * PPRT)
               (EQ (GETR V)
                   (QUOTE BE)))
        (HOLD (GETR SUBJ))
        (SETR SUBJ (BUILDQ (NP (PRO SOMEONE))))
        (SETR AGFLAG T)
        (SETR V *)
        (TO Q3/))
   (CAT V (AND (GETF * PPRT)
               (EQ (GETR V)
                   (QUOTE HAVE)))
        (SETR TNS (APPEND (GETR TNS)
                          (QUOTE (PERFECT))))
        (SETR V *)
        (TO Q3/))
   (PUSH NP/ (TRANS (GETR V))
        (SETR OBJ *)
        (TO Q4/))
   (VIR NP (TRANS (GETR V))
        (SETR OBJ *)
        (TO Q4/))
   (POP (BUILDQ (S + + (TNS +) (VP (V +)))
                TYPE SUBJ TNS V)
       (INTRANS (GETR V))))
```

```
(Q4/
   (WRD BY (GETR AGFLAG)
        (SETR AGFLAG NIL)
        (TO Q7/))
   (WRD TO (S-TRANS (GETR V))
        (TO Q5/))
   (POP (BUILDQ (S + + (TNS +) (VP (V +) +))
                TYPE SUBJ TNS V OBJ)
       T))
(Q5/
   (PUSH VP/ T
        (SENDR SUBJ (GETR OBJ))
        (SENDR TNS (GETR TNS))
        (SENDRQ TYPE DCL)
        (SETR OBJ *)
        (TO Q6/)))
(Q6/
   (WRD BY (GETR AGFLAG)
        (SETR AGFLAG NIL)
        (TO Q7/))
   (POP (BUILDQ (S + + (TNS +) (VP (V +) +))
                TYPE SUBJ TNS V OBJ)
       T))
(Q7/
   (PUSH NP/ T
        (SETR SUBJ *)
        (TO Q6/)))
(VP/
   (CAT V (GETF * UNTENSED)
        (SETR V *)
        (TO Q3/)))
(NP/
   (CAT DET T
        (SETR DET *)
        (TO NP/1))
   (CAT NPR T
        (SETR NPR *)
        (TO NP/3)))
(NP/1
   (CAT ADJ T
        (ADDL ADJS *)
        (TO NP/1))
   (CAT N T
        (SETR N *)
        (TO NP/2)))
(NP/2
   (POP (BUILDQ (NP (DET +) (ADJ +) (N +))
                DET ADJS N)
       T))
(NP/3
   (POP (BUILDQ (NP (NPR +))
                NPR)
       T))
)
```

Version I

```
(PARSER
  (LAMBDA (ACF)
    (PROG (STATE NODE STACK REGS HOLD * LEX)
```

The current status of the machine is kept in five global variables; (1) STATE, the state/arc in the grammar, (2) NODE, the pointer into the input, (3) REGS, the list of register name-value pairs, (4) STACK, the return stack, and (5) HOLD, the hold list. Putting the machine into a given configuration involves assigning values of these five variables.

```
      SPREAD-ACF
          (STATE←(CF.STATE ACF))
          (REGS←(CF.REGS ACF))
          (STACK←(CF.STACK ACF))
          (HOLD←(CF.HOLD ACF))
          (NODE←(CF.NODE ACF))
          (LEX←(EDGE.WORD (FIRST.EDGE NODE)))
```

BRANCH dispatches control to the label specified by STATE. This is the method of executing an arc.

```
      EVALARC
          (BRANCH STATE SUCCESS DETOUR S/ S/-2 S/-2-PUSH Q1/
                  Q1/-1-PUSH Q2/ Q3/ Q3/-2 Q3/-3 Q3/-4 Q3/-5
                  Q3/-3-PUSH Q4/ Q4/-2 Q4/-3 Q5/ Q5/-1-PUSH Q6/
                  Q6/-2 Q7/ Q7/-1-PUSH VP/ NP/
                  NP/-2 NP/1 NP/1-2 NP/2 NP/3)
```

SUCCESS checks to make sure all of the input has been processed. If not it detours.

```
      SUCCESS
          (if (EMPTYP.NODE NODE)
              then (RETURN *)
            else (GO DETOUR))
```

DETOUR decides which alternative to try next. In this case the alternatives list is a stack.

```
      DETOUR
          (if ALTS
              then ACF←(ALTS.FIRST)
                   (ALTS.BUTFIRST)
                   (GO SPREAD-ACF)
            else (RETURN (FAILURE)))
```

This is the beginning of the code which is compiled from the arcs. The first arc of each state has a label which is the same as the state name in the ATN. The other arcs have a label which is the state name followed by "-" and the arc number. Labels which end in "-PUSH" indicate the actions and termination action of PUSH arcs.

```
      S/  (if (ARCCAT AUX)
              then (ALTARC S/-2)
                   (SETR 'V *)
                   (SETR 'TNS <(GETF * 'TENSE)>)
                   (SETRQ TYPE Q)
                   (DOTO Q1/)
                   (GO Q1/))
```

```
S/-2(DOPUSH NP/ S/-2-PUSH)
    (GO NP/)
S/-2-PUSH
    (SETR 'SUBJ *)
    (SETRQ TYPE DCL)
    (DOPTO Q2/)
    (GO Q2/)
Q1/ (DOPUSH NP/ Q1/-1-PUSH)
    (GO NP/)
Q1/-1-PUSH
    (SETR 'SUBJ *)
    (DOPTO Q3/)
    (GO Q3/)
Q2/ (if (ARCCAT V)
        then (SETR 'V *)
             (SETR 'TNS <(GETF * 'TENSE)>)
             (DOTO Q3/)
             (GO Q3/))
     (GO DETOUR)
Q3/ (while (ARCCAT V) and (GETF * 'PPRT)
                      and (GETR V)='BE
       do (ALTARC Q3/-2)
          (HOLD (GETR SUBJ))
          (SETR 'SUBJ (BUILDQ (NP (PRO SOMEONE))))
          (SETR 'AGFLAG T)
          (SETR 'V *)
          (DOTO Q3/))
Q3/-2
    (if (ARCCAT V) and (GETF * 'PPRT)
                   and (GETR V)='HAVE
        then (ALTARC Q3/-3)
             (SETR 'TNS <! (GETR TNS) ! '(PERFECT)>)
             (SETR 'V *)
             (DOTO Q3/)
             (GO Q3/))
Q3/-3
    (if (TRANS (GETR V))
        then (ALTARC Q3/-4)
             (DOPUSH NP/ Q3/-3-PUSH)
             (GO NP/))
Q3/-4
          (if (HOLDSCAN HOLD 'NP '(TRANS (GETR V)))
        then (ALTARC Q3/-5)
             (PREVIRACTS)
             (SETR 'OBJ *)
             (DOVIRTO Q4/)
             (GO Q4/))
Q3/-5
    (if (INTRANS (GETR V))
        then (DOPOP (BUILDQ (S + +(TNS +) (VP (V +)))
                            TYPE SUBJ TNS V))
             (GO EVALARC))
    (GO DETOUR)
Q3/-3-PUSH
    (SETR 'OBJ *)
    (DOPTO Q4/)
    (GO Q4/)
Q4/ (if (ARCWRD BY) and (GETR AGFLAG)
        then (ALTARC Q4/-2)
             (SETR 'AGFLAG NIL)
             (DOTO Q7/)
             (GO Q7/))
    Q4/-2
        (if (ARCWRD TO) and (S-TRANS (GETR V))
            then (ALTARC Q4/-3)
                 (DOTO Q5/)
                 (GO Q5/))
    Q4/-3
        (DOPOP (BUILDQ (S + +(TNS +) (VP (V +)+))
                       TYPE SUBJ TNS V OBJ))
        (GO EVALARC)
    Q5/ (SENDR 'SUBJ (GETR OBJ))
        (SENDR 'TNS (GETR TNS))
        (SENDRQ TYPE DCL)
        (DOPUSH VP/ Q5/-1-PUSH)
        (SREGS NIL)
        (GO VP/)
    Q5/-1-PUSH
        (SETR 'OBJ *)
        (DOPTO Q6/)
        (GO Q6/)
    Q6/ (if (ARCWRD BY) and (GETR AGFLAG)
            then (ALTARC Q6/-2)
                 (SETR 'AGFLAG NIL)
                 (DOTO Q7/)
                 (GO Q7/))
    Q6/-2
        (DOPOP (BUILDQ (S + +(TNS +)
                            (VP (V +)+))
                       TYPE SUBJ TNS V OBJ))
        (GO EVALARC)
    Q7/ (DOPUSH NP/ Q7/-1-PUSH)
        (GO NP/)
    Q7/-1-PUSH
        (SETR 'SUBJ *)
        (DOPTO Q6/)
        (GO Q6/)
    VP/ (if (ARCCAT V) and (GETF * 'UNTENSED)
            then (SETR 'V *)
                 (DOTO Q3/)
                 (GO Q3/))
        (GO DETOUR)
    NP/ (if (ARCCAT DET)
            then (ALTARC NP/-2)
                 (SETR 'DET *)
                 (DOTO NP/1)
                 (GO NP/1))
    NP/-2
        (if (ARCCAT NPR)
            then (SETR 'NPR *)
                 (DOTO NP/3)
                 (GO NP/3))
        (GO DETOUR)
    NP/1(while (ARCCAT ADJ) do (ALTARC NP/1-2)
                               (ADDL 'ADJS *)
                               (DOTO NP/1))
    NP/1-2
        (if (ARCCAT N)
            then (SETR 'N *)
                 (DOTO NP/2)
                 (GO NP/2))
        (GO DETOUR)
    NP/2(DOPOP (BUILDQ (NP (DET +)
                           (ADJ +)
                           (N +))
                       DET ADJS N))
        (GO EVALARC)
    NP/3(DOPOP (BUILDQ (NP (NPR +))
                       NPR))
        (GO EVALARC))))
)
```

Version II

```
(PARSER
  (LAMBDA (ACF)
    (PROG (STATE NODE STACK REGS FEATS HOLD * LEX SREGS
                SFEATS FEATURES [illegible]MP)
```

If the function is called with an argument of ´GO, it looks for another parse. This allows the user to get out more than the first parse.

```
     (if ACF=´GO
         then (GO DETOUR))
```

The current status of the machine is kept in five global variables: (1) STATE, the state/arc in the grammar, (2) NODE, the pointer into the input, (3) REGS, the list of register name-value pairs, (4) STACK, the return stack, and (5) HOLD, the hold list. Putting the machine into a given configuration involves assigning values to these five variables.

```
  SPREAD-ACF
      (CHANGESTATE (CF.STATE ACF))
      (REGS←(CF.REGS ACF))
      (FEATS←(CF.FEATS ACF))
      (STACK←(CF.STACK ACF))
      (HOLD←(CF.HOLD ACF))
      (LEX←(EDGE.WORD (FIRST.EDGE NODE←(CF.NODE ACF))))
```

TRACEALTSTART is one of the tracing functions provided to allow the user to follow the operations of the parser. The others are TRACEARC and ABORT. None of these result in any code when a fast version of the parser is produced.

```
      (TRACEALTSTART)
      (GO EVALARC)
  NEXTLEX
```

If the current node has more than one lexical interpretation (BUTFIRST.EDGE), the code sets NODE to try the next one.

```
      (if (BUTFIRST.EDGE NODE)
          then LEX←(EDGE.WORD (FIRST.EDGE
                                  NODE←(BUTFIRST.EDGE
                                            NODE)))
               (GO EVALARC))
```

BRANCH dispatches control to the label specified by STATE.

```
  EVALARC
      (BRANCH STATE SUCCESS DETOUR S/ S/-1-CONT S/-2
              S/-1-CAT  S/-2-PUSH Q1/ Q1/-1-PUSH Q2/
              Q2/-1-CONT Q2/-1-CAT Q3/ Q3/-1-CONT
              Q3/-2 Q3/-2-CONT Q3/-3 Q3/-4 Q3/-5
              Q3/-1-CAT Q3/-2-CAT Q3/-3-PUSH Q4/ Q4/-2 Q4/-3
              Q5/ Q5/-1-PUSH Q6/ Q6/-2 Q7/ Q7/-1-PUSH
              VP/ VP/-1-CONT VP/-1-CAT NP/ NP/-1-CONT
              NP/-2 NP/-2-CONT NP/-1-CAT NP/-2-CAT
              NP/1 NP/1-1-CONT NP/1-2 NP/1-2-CONT
              NP/1-1-CAT NP/1-2-CAT NP/2 NP/3)
  SUCCESS
      (RETURN NODE)
```

DETOUR chooses an alternative from the ALTS list. In this version the ALTS list is a stack. The detouring mechanism could be changed by redefining ALTS.FIRST and ALTS.BUTFIRST. If there are no more alternatives, the first alternative from the list or SUSPENDED alts is taken. The suspended alternatives are maintained in order by weight.

```
ABORT
    (ABORT)                         ABORT is a tracing function.
DETOUR
    (if ALTS
        then ACF←(ALTS.FIRST)
             (ALTS.BUTFIRST)
             (GO SPREAD-ACF)
      elseif SUSPENDEDALTS
        then ACF←(SUSPEND.POP)
             (GO SPREAD-ACF)
      else  (RETURN (FAILURE)))
S/  (if (ARCCAT AUX)
      else (GO S/-2))
    (ALTARC S/-2)
    (TRACEARC CAT AUX S/-1)
S/-1-CONT
    (ALTCAT S/-1-CAT)
    (SETR 'V *)
    (SETR 'TNS <(GETF * 'TENSE)>)
    (SETRQ TYPE Q)
    (DOTO Q1/)
    (GO Q1/)
S/-2(if (STRINGLEFTP)
        then (NEXTLEXALT S/)
             (TRACEARC PUSH NIL S/-2)
             (DOPUSH NP/ S/-2-PUSH)
             (GO DETOUR))
    (CHANGESTATEQ S/)
    (GO NEXTLEX)
S/-1-CAT
    (ARCCAT AUX)
    (TRACEARC ALTCAT AUX S/-1)
    (GO S/-1-CONT)
S/-2-PUSH
    (SPREAD/WFS)
    (SETR 'SUBJ *)
    (SETRQ TYPE DCL)
    (DOPTO Q2/)
    (GO Q2/)
Q1/ (if (STRINGLEFTP)
        then (NEXTLEXALT Q1/)
             (TRACEARC PUSH NIL Q1/-1)
             (DOPUSH NP/ Q1/-1-PUSH)
             (GO DETOUR))
    (CHANGESTATEQ Q1/)
    (GO NEXTLEX)
Q1/-1-PUSH
    (SPREAD/WFS)
    (SETR 'SUBJ *)
    (DOPTO Q3/)
    (GO Q3/)
Q2/ (if (ARCCAT V)
      else (CHANGESTATEQ Q2/)
           (GO NEXTLEX))
    (NEXTLEXALT Q2/)
    (TRACEARC CAT V Q2/-1)
```

```
Q2/-1-CONT
    (ALTCAT Q2/-1-CAT)
    (SETR 'V *)
    (SETR 'TNS <(GETF * 'TENSE)>)
    (DOTO Q3/)
    (GO Q3/)
Q2/-1-CAT
    (ARCCAT V)
    (TRACEARC ALTCAT V Q2/-1)
    (GO Q2/-1-CONT)
Q3/ (if (ARCCAT V)
      else (GO Q3/-2))
    (ALTARC Q3/-2)
    (TRACEARC CAT V Q3/-1)
Q3/-1-CONT
    (ALTCAT Q3/-1-CAT)
    (if ~((GETF * 'PPRT) and (GETR V)='BE)
        then (GO ABORT))
    (HOLD (GETR SUBJ))
    (SETR 'SUBJ (BUILDQ (NP (PRO SOMEONE))))
    (SETR 'AGFLAG T)
    (SETR 'V *)
    (DOTO Q3/)
    (GO Q3/)
Q3/-2
    (if (ARCCAT V)
      else (GO Q3/-3))
    (ALTARC Q3/-3)
    (TRACEARC CAT V Q3/-2)
Q3/-2-CONT
    (ALTCAT Q3/-2-CAT)
    (if ~((GETF * 'PPRT) and (GETR V)='HAVE)
        then (GO ABORT))
    (SETR 'TNS <! (GETR TNS) ! '
                  (PERFECT)>)
    (SETR 'V *)
    (DOTO Q3/)
    (GO Q3/)
Q3/-3
    (if (STRINGLEFTP) and (TRANS (GETR V))
         then (ALTARC Q3/-4)
              (TRACEARC PUSH NIL Q3/-3)
              (DOPUSH NP/ Q3/-3-PUSH)
              (GO DETOUR))
Q3/-4
    (if TEMP (HOLDSCAN HOLD 'NP '(TRANS (GETR V)))
         then (ALTARC Q3/-5)
              (TRACEARC VIR NP Q3/-4)
              (PREVIRACTS)
              (SETR 'OBJ *)
              (DOVIRTO Q4/)
              (GO Q4/))
Q3/-5
    (if (INTRANS (GETR V))
         then (NEXTLEXALT Q3/)
              (TRACEARC POP NIL Q3/-5)
              (DOPOP (BUILDQ (S + +(TNS +)
                                    (VP (V +)))
                             TYPE SUBJ TNS V)
                     (GETR POPFEATS))
              (GO DETOUR))
    (CHANGESTATEQ Q3/)
    (GO NEXTLEX)
```

```
Q3/-1-CAT
    (ARCCAT V)
    (TRACEARC ALTCAT V Q3/-1)
    (GO Q3/-1-CONT)
Q3/-2-CAT
    (ARCCAT V)
    (TRACEARC ALTCAT V Q3/-2)
    (GO Q3/-2-CONT)
Q3/-3-PUSH
    (SPREAD/WFS)
    (SETR 'OBJ *)
    (DOPTO Q4/)
    (GO Q4/)
Q4/ (if (ARCWRD BY) and (GETR AGFLAG)
        then (ALTARC Q4/-2)
             (TRACEARC WRD BY Q4/-1)
             (SETR 'AGFLAG NIL)
             (DOTO Q7/)
             (GO Q7/))
Q4/-2
    (if (ARCWRD TO) and (S-TRANS (GETR V))
        then (ALTARC Q4/-3)
             (TRACEARC WRD TO Q4/-2)
             (DOTO Q5/)
             (GO Q5/))
Q4/-3
    (NEXTLEXALT Q4/)
    (TRACEARC POP NIL Q4/-3)
    (DOPOP (BUILDQ (S + +(TNS +)
                       (VP (V +)+))
                   TYPE SUBJ TNS V OBJ)
           (GETR POPFEATS))
    (GO DETOUR)
Q5/ (if (STRINGLEFTP)
        then (NEXTLEXALT Q5/)
             (TRACEARC PUSH NIL Q5/-1)
             (SENDR 'SUBJ (GETR OBJ))
             (SENDR 'TNS (GETR TNS))
             (SENDRQ TYPE DCL)
             (DOPUSH VP/ Q5/-1-PUSH)
             SREGS←NIL
             SFEATS←NIL
             (GO DETOUR))
    (CHANGESTATEQ Q5/)
    (GO NEXTLEX)
Q5/-1-PUSH
    (SPREAD/WFS)
    (SETR 'OBJ *)
    (DOPTO Q6/)
    (GO Q6/)
Q6/ (if (ARCWRD BY) and (GETR AGFLAG)
        then (ALTARC Q6/-2)
             (TRACEARC WRD BY Q6/-1)
             (SETR 'AGFLAG NIL)
             (DOTO Q7/)
             (GO Q7/))
Q6/-2
    (NEXTLEXALT Q6/)
    (TRACEARC POP NIL Q6/-2)
    (DOPOP (BUILDQ (S + +(TNS +)
                       (VP (V +)+))
                   TYPE SUBJ TNS V OBJ)
           (GETR POPFEATS))
    (GO DETOUR)
```

```
Q7/ (if (STRINGLEFTP)
         then (NEXTLEXALT Q7/)
              (TRACEARC PUSH NIL Q7/-1)
              (DOPUSH NP/ Q7/-1-PUSH)
              (GO DETOUR))
    (CHANGESTATEQ Q7/)
    (GO NEXTLEX)
Q7/-1-PUSH
    (SPREAD/WFS)
    (SETR 'SUBJ *)
    (DOPTO Q6/)
    (GO Q6/)
VP/ (if (ARCCAT V)
      else (CHANGESTATEQ VP/)
           (GO NEXTLEX))
    (NEXTLEXALT VP/)
    (TRACEARC CAT V VP/-1)
VP/-1-CONT
    (ALTCAT VP/-1-CAT)
    (if ~(GETF * 'UNTENSED)
        then (GO ABORT))
    (SETR 'V *)
    (DOTO Q3/)
    (GO Q3/)
VP/-1-CAT
    (ARCCAT V)
    (TRACEARC ALTCAT V VP/-1)
    (GO VP/-1-CONT)
NP/ (if (ARCCAT DET)
      else (GO NP/-2))
    (ALTARC NP/-2)
    (TRACEARC CAT DET NP/-1)
NP/-1-CONT
    (ALTCAT NP/-1-CAT)
    (SETR 'DET *)
    (DOTO NP/1)
    (GO NP/1)
NP/-2
    (if (ARCCAT NPR)
      else (CHANGESTATEQ NP/)
           (GO NEXTLEX))
    (NEXTLEXALT NP/)
    (TRACEARC CAT NPR NP/-2)
NP/-2-CONT
    (ALTCAT NP/-2-CAT)
    (SETR 'NPR *)
    (DOTO NP/3)
    (GO NP/3)
NP/-1-CAT
    (ARCCAT DET)
    (TRACEARC ALTCAT DET NP/-1)
    (GO NP/-1-CONT)
NP/-2-CAT
    (ARCCAT NPR)
    (TRACEARC ALTCAT NPR NP/-2)
    (GO NP/-2-CONT)
NP/1(if (ARCCAT ADJ)
      else (GO NP/1-2))
    (ALTARC NP/1-2)
    (TRACEARC CAT ADJ NP/1-1)
NP/1-1-CONT
    (ALTCAT NP/1-1-CAT)
    (ADDL 'ADJS *)
    (DOTO NP/1)
    (GO NP/1)
```

```
      NP/1-2
          (if (ARCCAT N)
            else (CHANGESTATEQ NP/1)
                 (GC NEXTLEX))
          (NEXTLEXALT NP/1)
          (TRACEARC CAT N NP/1-2)
      NP/1-2-CONT
          (ALTCAT NP/1-2-CAT)
          (SETR 'N *)
          (DOTO NP/2)
          (GO NP/2)
      NP/1-1-CAT
          (ARCCAT ADJ)
          (TRACEARC ALTCAT ADJ NP/1-1)
          (GO NP/1-1-CONT)
      NP/1-2-CAT
          (ARCCAT N)
          (TRACEARC ALTCAT N NP/1-2)
          (GO NP/1-2-CONT)
      NP/2(NEXTLEXALT NP/2)
          (TRACEARC POP NIL NP/2-1)
          (DOPOP (BUILDQ (NP (DET +)
                             (ADJ +)
                             (N +))
                          DET ADJS N)
                 (GETR POPFEATS))
          (GO DETOUR)
      NP/3(NEXTLEXALT NP/3)
          (TRACEARC POP NIL NP/3-1)
          (DOPOP (BUILDQ (NP (NPR +))
                          NPR)
                 (GETR POPFEATS))
          (GO DETOUR))))
)
```

Trace of Version I Parsing a Sentence

```
PARSE((JOHN WAS BELIEVED TO HAVE BEEN SHOT BY FRED))

Starting alternative 0
At arc S/
Node = (((JOHN NPR (&)) ((WAS V & AUX &) (& &))))
```

The sentence is converted into a chart format. The chart contains information about the possible parts of speech of each word. Notice that "was" can be either a verb (V) or an auxiliary verb (AUX). (An "&" is used to indicate a further structure.)

```
Taking PUSH arc S/-2
```

The trace indicates the arc type and its location in the grammar. No alternative is stored because S/-2 is the last arc in the state S/ and there are no lexical alternatives.

```
PUSHing for NP/
 Taking CAT NPR arc NP/-2
    Setting NPR to JOHN
```

The trace also indicates where registers get set.

```
Entering state NP/3
 Node = (((WAS V (&) AUX (&)) ((BELIEVED V &) (& &))))
 Taking POP arc NP/3-1
 Trying to POP
(Continuing arc S/-2-PUSH)
   Setting SUBJ to (NP (NPR JOHN))
   Setting TYPE to DCL

Entering state Q2/
Node = (((WAS V (&) AUX (&)) ((BELIEVED V &) (& &))))
Taking CAT V arc Q2/-1
   Setting V to BE
   Setting TNS to (PAST)

Entering state Q3/
Node = (((BELIEVED V (&)) ((TO PREP &) (& &))))
```

The alternative configuration to try the second arc leaving Q3/ (Q3/2) is created and saved after the test has succeeded on the first arc but before the arc is taken. This is alt 2 because configuration 1 was created during the earlier PUSH arc (i.e. the number is a configuration number).

```
      Storing alt 2 for arc Q3/-2
Taking CAT V arc Q3/-1
HOLDing (NP (NPR JOHN))
   Setting SUBJ to (NP (PRO SOMEONE))
   Setting AGFLAG to T
   Setting V to BELIEVE

Entering state Q3/
Node = (((TO PREP (&)) ((HAVE V &) (& &))))
      Storing alt 3 for arc Q3/-4
Taking PUSH arc Q3/-3
PUSHing for NP/
 BLOCKED

Starting alternative 3
At arc Q3/-4
```

```
Node = (((TO PREP (&)) ((HAVE V &) (& &))))
      Storing alt 5 for arc Q3/-5
Taking VIR NP arc Q3/-4
(NP (NPR JOHN)) removed from HOLD list
   Setting OBJ to (NP (NPR JOHN))

Entering state Q4/
Node = (((TO PREP (&)) ((HAVE V &) (& &))))
      Storing alt 6 for arc Q4/-3
Taking WRD TO arc Q4/-2

Entering state Q5/
Node = (((HAVE V (&)) ((BEEN V &) (& &))))
Taking PUSH arc Q5/-1
   SENDing SUBJ value of (NP (NPR JOHN))
   SENDing TNS value of (PAST)
   SENDing TYPE value of DCL
PUSHing for VP/
 Taking CAT V arc VP/-1
    Setting V to HAVE

Entering state Q3/
 Node = (((BEEN V (&)) ((SHOT V &) (& &))))
       Storing alt 8 for arc Q3/-3
 Taking CAT V arc Q3/-2
    Setting TNS to (PAST PERFECT)
    Setting V to BE

Entering state Q3/
 Node = (((SHOT V (&)) ((BY PREP &) (& NIL))))
       Storing alt 9 for arc Q3/-2
 Taking CAT V arc Q3/-1
 HOLDing (NP (NPR JOHN))
    Setting SUBJ to (NP (PRO SOMEONE))
    Setting AGFLAG to T
    Setting V to SHOOT

Entering state Q3/
 Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
       Storing alt 10 for arc Q3/-4
 Taking PUSH arc Q3/-3
 PUSHing for NP/
  BLOCKED

Starting alternative 10
At arc Q3/-4
Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
       Storing alt 12 for arc Q3/-5
 Taking VIR NP arc Q3/-4
 (NP (NPR JOHN)) removed from HOLD list
    Setting OBJ to (NP (NPR JOHN))

Entering state Q4/
 Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
       Storing alt 13 for arc Q4/-2
 Taking WRD BY arc Q4/-1
    Setting AGFLAG to NIL

Entering state Q7/
 Node = (((FRED NPR (&)) NIL))
 Taking PUSH arc Q7/-1
 PUSHing for NP/
  Taking CAT NPR arc NP/-2
     Setting NPR to FRED
```

```
Entering state NP/3
  Node = (NIL)
  Taking POP arc NP/3-1
  Trying to POP
 (Continuing arc Q7/-1-PUSH)
    Setting SUBJ to (NP (NPR FRED))

Entering state Q6/
 Node = (NIL)
 Taking POP arc Q6/-2
 Trying to POP
(Continuing arc Q5/-1-PUSH)
   Setting OBJ to (S DCL (NP (NPR FRED))
                         (TNS (PAST PERFECT))
                         (VP
(V SHOOT) (NP (NPR JOHN))))

Entering state Q6/
Node = (NIL)
Taking POP arc Q6/-2
Trying to POP
Trying to SUCCEED

S DCL
  NP PRO SOMEONE
  TNS PAST
  VP V BELIEVE
     S DCL
       NP NPR FRED
       TNS PAST PERFECT
       VP V SHOOT
          NP NPR JOHN
```

One successful parse. Parser continues because it was being run in a mode which returns all possible parses.

```
Starting alternative 13
At arc Q4/-2
Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
 Taking POP arc Q4/-3
 Trying to POP
(Continuing arc Q5/-1-PUSH)
   Setting OBJ to (S DCL (NP (PRO SOMEONE))
                         (TNS (PAST PERFECT))
                         (VP
(V SHOOT) (NP (NPR JOHN))))

Entering state Q6/
Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
      Storing alt 15 for arc Q6/-2
Taking WRD BY arc Q6/-1
   Setting AGFLAG to NIL

Entering state Q7/
Node = (((FRED NPR (&)) NIL))
Taking PUSH arc Q7/-1
PUSHing for NP/
 Taking CAT NPR arc NP/-2
    Setting NPR to FRED

Entering state NP/3
 Node = (NIL)
 Taking POP arc NP/3-1
 Trying to POP
(Continuing arc Q7/-1-PUSH)
   Setting SUBJ to (NP (NPR FRED))
```

```
Entering state Q6/
Node = (NIL)
Taking POP arc Q6/-2
Trying to POP
Trying to SUCCEED

S DCL
  NP NPR FRED
  TNS PAST
  VP V BELIEVE                                              Second possible parse.
     S DCL
       NP PRO SOMEONE
       TNS PAST PERFECT
       VP V SHOOT
          NP NPR JOHN

Starting alternative 15
At arc Q6/-2
Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
Taking POP arc Q6/-2
Trying to POP
Trying to SUCCEED
BLOCKED

Starting alternative 12
At arc Q3/-5
Node = (((BY PREP (&)) ((FRED NPR &) NIL)))
 BLOCKED

Starting alternative 9
At arc Q3/-2
Node = (((SHOT V (&)) ((BY PREP &) (& NIL))))
 BLOCKED

Starting alternative 8
At arc Q3/-3
Node = (((BEEN V (&)) ((SHOT V &) (& &))))
 BLOCKED

Starting alternative 6
At arc Q4/-3
Node = (((TO PREP (&)) ((HAVE V &) (& &))))
Taking POP arc Q4/-3
Trying to POP
Trying to SUCCEED
BLOCKED

Starting alternative 5
At arc Q3/-5
Node = (((TO PREP (&)) ((HAVE V &) (& &))))
BLOCKED

Starting alternative 2
At arc Q3/-2
Node = (((BELIEVED V (&)) ((TO PREP &) (& &))))
BLOCKED
NIL
```

Appendix E

Grammar Compiler Declarations

Specification of Features

Some features of the general ATN parser require a good deal of bookkeeping. For example, SYSCONJ requires a parser to save the path that it takes through the grammar. This more than doubles the amount of storage overhead. To relieve the burden of those features, such as SYSCONJ, which increase the overhead, and which a particular application may not require, the user can specify which features his grammar uses. The compiler will then tailor the object code to those needs. The user specifications consist of a collection of flags which are set at compile time. A description of each flag together with its default setting is given below.

HOLDFLG: If the grammar does not use the HOLD facility, setting this flag to NIL will eliminate one field in a configuration. Default is T.

FEATURESFLG: If the grammar doesn't use the feature facility, setting this flag to NIL will eliminate one field in a configuration. Default is T.

WFSTFLG: If the grammar uses the well-formed substring feature, WFSTFLG should be non-NIL. Default is NIL.

ALTCATSFLG: If this flag is NIL, the compiler will not compile the ability to handle multiple interpretations of a word *within a single category*. If ALTCATSFLG is a list, it will compile this ability into those CAT arcs whose categories are members of the list. If T, it will compile this ability into all CAT arcs. Default is T.

SYSCONJFLG: If the grammar uses the LUNAR SYSCONJ conjunction-handling facility SYSCONJFLG should be non-NIL. Default is NIL. (SYSCONJ has not been implemented yet.)

STARTSTATE: This should be the start state of the grammar. Default value is S/.

NULLPUSHFLG: If NULLPUSHFLG is non-NIL, a PUSH arc will never be taken if there is no input left.[1] Default setting is T.

UNAMBIGUOUS-CHART: If the input chart is never ambiguous, setting this flag to a non-NIL value will avoid the checking for an alternative lexical interpretation. Default is NIL.

---

[1] This begins to legislate out PUSHes which do not use any of the inputs. In practical terms, this means that a PUSHed to network has to do more than just take constituents off the hold list. In theoretical terms, it closes one of the holes which may allow an ATN grammar to be undecidable.

Declarations for Arc Tests and Actions

The tests and actions on an arc can be arbitrary LISP expressions. To compile these function calls, the grammar compiler must know which arguments get evaluated. In general the grammar compiler gets this information from the same declarations about functions that the LISP compiler uses (NLAMA, NLAML, FNTYPE, etc.). In addition a facility is provided which allows the user to tell the grammar compiler how to compile the individual arguments to particular functions. Using this facility it is possible to write function calls in the grammar which implicitly QUOTE some of their arguments and evaluate others or even which call another function to decode their arguments. The compiler is told how to compile the arguments to a function by putting a specification as the value of the property GRAMMARARGINFO on the property list of the function name. The value of GRAMMARARGINFO property should be one of the following:

1) LAMBDA: the function evaluates all of its arguments. (This is the default case.)

2) NLAMBDA: the function doesn't evaluate any of its arguments. This can also be done by putting the function on either of the lists NLAMA or NLAML (see INTERLISP compiler).

3) A list which specifies how each argument should be treated. Each element of the list can be:

   1) E or NIL - This argument position will be evaluated. This is the usual case where the action expects its argument to be evaluated and tells the grammar compiler to scan the argument for embedded calls.

   2) QUOTE - This argument is embedded in QUOTE. This provides a convenient way of automatically quoting certain argument positions in a function call.

   3) * - The argument is not compiled by the grammar compiler but is merely copied. Note: Arguments which occur in this position should not have any embedded functions as these will not be scanned by the compiler.

   4) Any other atom - The atom is the name of a function which when APPLYed to the argument returns the compiled form.

Examples: The grammar function SETR which sets the value of a register could be compiled by having a GRAMMARARGINFO property of (QUOTE E).[2] The arc action (SETR ANAPHORFLG T) would compile into (SETR (QUOTE ANAPHORFLG) T). SETR is defined as a LAMBDA function (i.e. the interpreter evaluates its arguments) which avoids the explicit call to EVAL which results from having SETR be a NLAMBDA function (i.e. the interpreter doesn't evaluate its arguments).

---

[2] SETR is, in fact, recognized specially by the grammar compiler so that it can keep track of the registers which are used in the grammar.

In the LUNAR grammar, many of the arc functions use EVALLOC to evaluate one or more of their arguments. EVALLOC has three options: (1) if its argument is "*" or NIL, it gets the value of the current thing - *; (2) if the argument is atomic, it is a register whose value is retrieved; and (3) if the argument is a list, it is evaluated. This allows the grammar to be clearer and less cluttered with predictable function calls. To accomplish the same results using the compiler, a version of EVALLOC (CEVALLOC) is provided which returns the form for the decoded argument. The functions which use it are then given GRAMMARARGINFO property of CEVALLOC for those argument positions which need decoding. This means that the decoding process takes place once at compile time instead of each time the arc is tried. For example, in the LUNAR grammar the function MARKER has a GRAMMARARGINFO property of (CEVALLOC QUOTE). This allows the grammar to have (MARKER N MASS) as an action which compiles into (MARKER (GETR N) (QUOTE MASS)) and avoids an explicit call to EVAL by MARKER. Notice that by using this technique, the grammar writer can easily specify default arguments to actions in his grammar (at very little computational cost) and greatly improve the readability of the grammar.

Appendix F

Debugging Features

Since the compiler transforms the grammar into a program, the grammar writer can use the debugging features of the object language to aid in debugging his grammar. These should, of course, be augmented by some features particular to grammars, but these are best integrated into an existing framework. The following section describes a collection of grammar debugging tools that have been integrated into the INTERLISP system.

The debugging facilities can be grouped into two major categories; tracing and breaking. The trace will show all grammar transitions and register-changing operations. In debugging mode, the system will even keep a complete history of the parse so that the user can back up. In addition, the user has the ability to stop the parser at the end of each line of the trace in order to look around in and/or change the current environment.

Tracing

The trace package causes the functions in the object language program to print out what they are doing. There are three types of actions which may be included in the trace: (1) arc transitions, storing of alternatives and hold list operations; (2) setting of registers; and (3) sending of registers to a PUSH configuration. The latter two of these can be turned off independently. In addition, the debugging system allows the user to trace to a disk file and not to TTY. (If the user wants both TTY and disk copies, he can use the INTERLISP DRIBBLE facility.)

Breaks

The break package allows the user to stop the parser at any time; check the states of the current configuration, or any of the alternative or previously blocked configurations; or backup to previous points in the parse to examine more closely the path taken. The break package exec is

BREAK1 (the LISP Break executive) augmented with some special functions and BREAKMACROS. Since the user is talking to BREAK1, he can use any of the LISP break commands or execute any LISP functions as well as the special commands described below. He can also use the special commands while inside of a break caused by having broken one of his functions or typing Control-H or Control-B.

How to Get into a Break

Whenever the trace package prints a line of tracing information, and the variable PAUSEFLAG has a non-NIL value, the trace package will wait for the user to indicate whether to continue or break. A break is caused by typing PAUSECHAR (initially ","). Continuation is caused by typing CONTINUECHAR (initially "."). All other characters are ignored. If PAUSECHAR is typed, BREAK1 is entered. The parsing is resumed by using one of the Break exiting commands, or by using one of the special commands described below. Note: "." is equivalent to the Break command "OK".

Grammar Break Commands

Printing Out Parsing Information:

The following commands and functions are provided to print out information associated with a configuration.

1) CF - a Break command which prints out the present status of the currently active configuration.

2) PPCF(n) - prints out the status of configuration number n.

Note: Both CF and PPCF only print non-NIL information about a configuration. Also PRINTLEVEL is set to 4 when debugging. It can be reset to a higher (or lower) number if the user wants more (or less) information printed.

3) PT - a Break command which tree prints (PPTT - see below) the current structure (*). This is most useful after a POP to examine the structure which was POPped.

4) PPTT(x) - prints the structure x in a tree format without parentheses.

5) CFARRAYDUMP(ST END) - dumps the contents of the configuration array from configuration number ST to configuration number END. If ST is NIL, 0 is used. If END is NIL, the largest configuration FREECF#, is used.

The following commands are used to change the flow of control of the parser while debugging. In order to use AGAIN or BACKUP, the parser must be run in PATH mode, which saves a new configuration each time an arc is taken.

1) AGAIN - a break command which restarts the current configuration, i.e. goes back to the most recent arc transition and starts again. The effect is to redo the current arc. If the user discovers that this did not back up far enough, he can use the command BACKUP.

2) BACKUP - a break command which restarts the configuration which led to the current one. BACKUP may be invoked successively to back up more than one arc transition.

3) ABORT - a break command which ABORTs the current configuration. The next active configuration will be taken from the ALTS list.

Note: AGAIN and BACKUP are useful if an arc is taken (or not taken) when it should not have been (or should have been). The predicates or functions involved in the offending arc test can be broken (using the LISP function BREAK) and then AGAIN or BACKUP can be called to redo the arc.

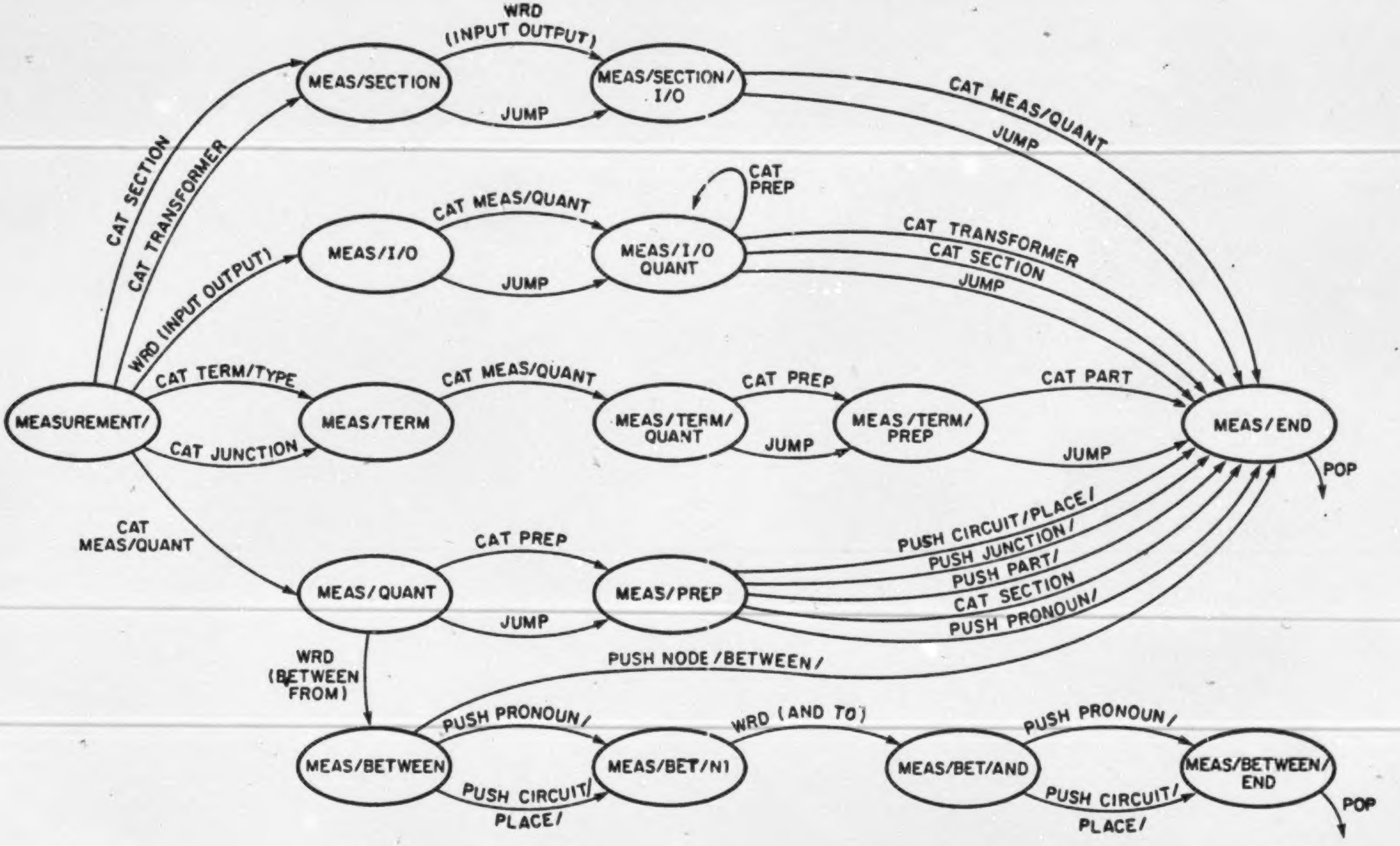
4) FIRE(n) - aborts the current configuration and starts the configuration n. If n is on the ALTS list, the ALTS list is POPped to the configuration before n.

5) PARSER(n) - recursively invokes the parser on configuration n. This provides a way of exploring one of the configurations on the ALTS list or returning to a (much) earlier configuration. Note: After PARSER returns, the user is still in the same place with respect to the current parse (except that he may have fewer configurations left, his alternative lists may have been altered and his WFST may contain more entries.)

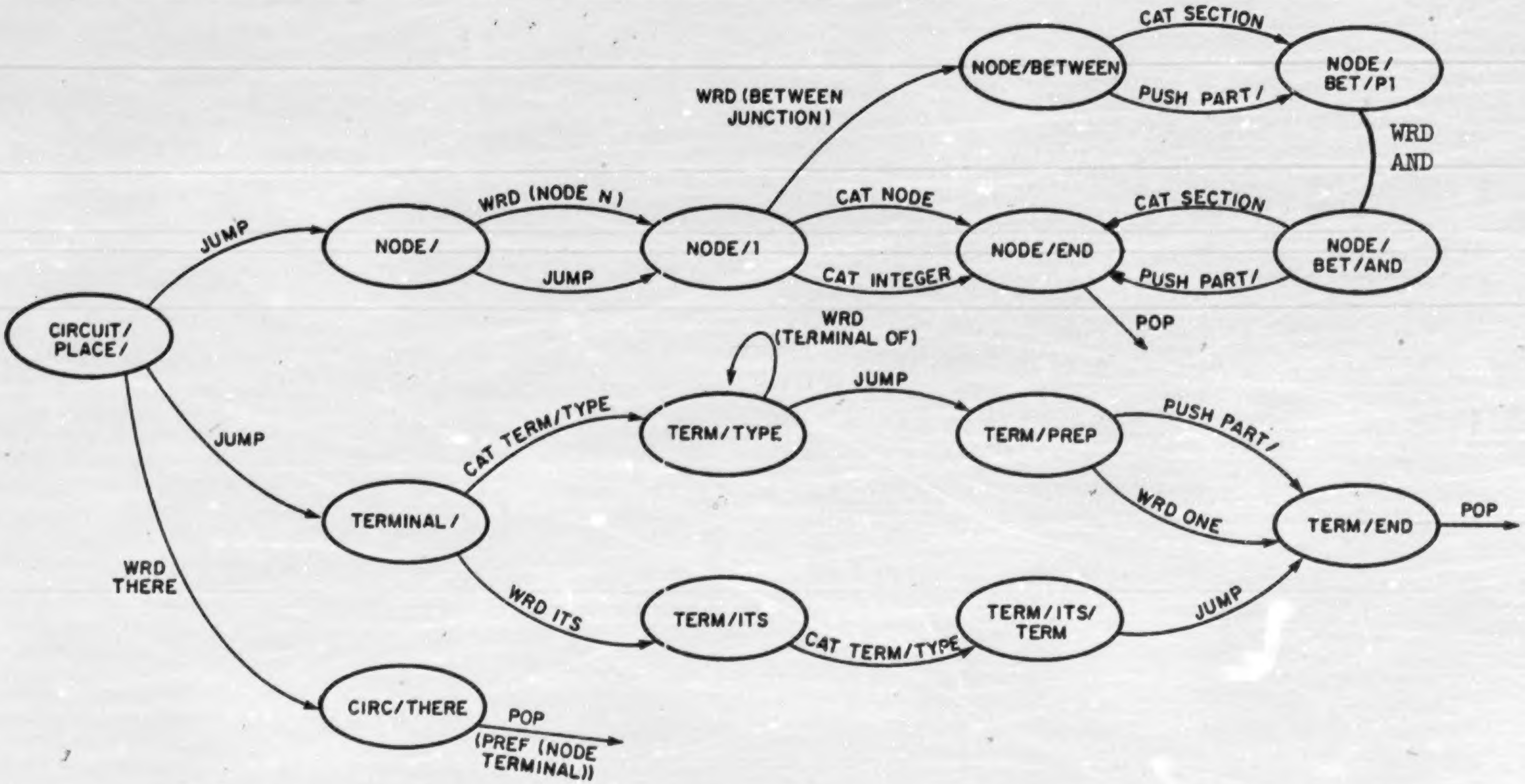
Appendix G

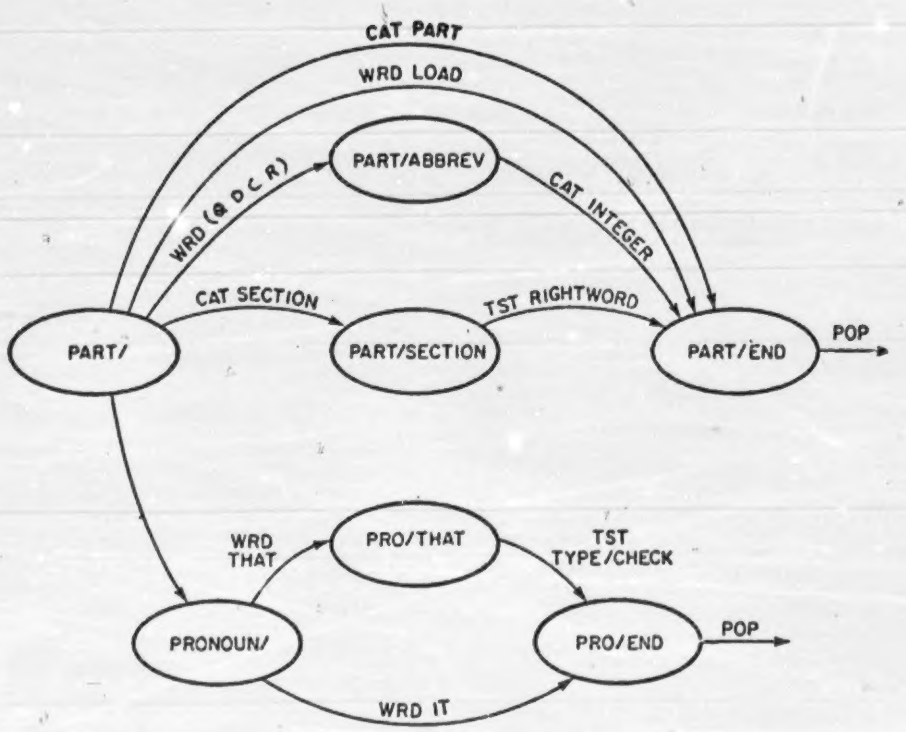
ATN Description of Part of the SOPHIE Semantic Grammar

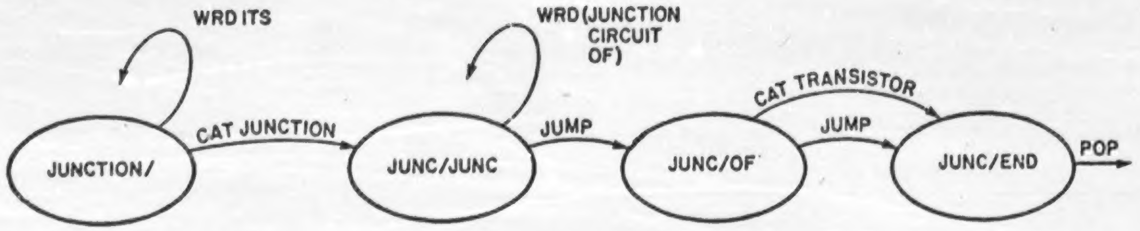
This appendix gives an ATN description of the same subset of the language as presented in Appendix A. Of the 24 rules listed in Appendix A, 15 became "syntactic" categories, 3 were incorporated into other networks and 6 remained non-terminals. The first section presents the ATN in its graphic form. The second section presents the ATN as it is input to the compiler.


WRD
(INPUT OUTPUT)
MEAS/SECTION
JUMP
MEAS/SECTION/
I/O
CAT MEAS/QUANT
JUMP
CAT SECTION
CAT TRANSFORMER
CAT
PREP
CAT MEAS/QUANT
MEAS/I/O
JUMP
MEAS/I/O
QUANT
CAT TRANSFORMER
CAT SECTION
JUMP
WRD (INPUT OUTPUT)
CAT TERM/TYPE
CAT MEAS/QUANT
CAT PREP
CAT PART
MEASUREMENT/
MEAS/TERM
MEAS/TERM/
QUANT
MEAS/TERM/
PREP
MEAS/END
CAT JUNCTION
JUMP
JUMP
POP
CAT
MEAS/QUANT
PUSH CIRCUIT/PLACE/
PUSH JUNCTION/
PUSH PART/
CAT SECTION
PUSH PRONOUN/
CAT PREP
MEAS/QUANT
MEAS/PREP
JUMP
WRD
(BETWEEN
FROM)
PUSH NODE/BETWEEN/
PUSH PRONOUN/
WRD (AND TO)
PUSH PRONOUN/
MEAS/BETWEEN
MEAS/BET/N1
MEAS/BET/AND
MEAS/BETWEEN/
END
PUSH CIRCUIT/
PLACE/
PUSH CIRCUIT/
PLACE/
POP

Graphic Form of Semantic ATN


CAT SECTION
NODE/BETWEEN
NODE/
BET/P1
PUSH PART/
WRD (BETWEEN
JUNCTION)
WRD
AND
WRD (NODE N)
CAT NODE
CAT SECTION
JUMP
NODE/
NODE/1
NODE/END
NODE/
BET/AND
JUMP
CAT INTEGER
PUSH PART/
CIRCUIT/
PLACE/
POP
WRD
(TERMINAL OF)
JUMP
CAT TERM/TYPE
TERM/TYPE
TERM/PREP
PUSH PART/
JUMP
WRD ONE
TERMINAL/
TERM/END
POP
WRD
THERE
WRD ITS
TERM/ITS
TERM/ITS/
TERM
JUMP
CAT TERM/TYPE
CIRC/THERE
POP
(PREF (NODE
TERMINAL))


CAT PART
WRD LOAD
PART/ABBREV
WRD (Q D C R)
CAT INTEGER
CAT SECTION
TST RIGHTWORD
PART/
PART/SECTION
PART/END
POP
WRD THAT
PRO/THAT
TST TYPE/CHECK
PRONOUN/
PRO/END
POP
WRD IT


WRD ITS
WRD(JUNCTION CIRCUIT OF)
CAT TRANSISTOR
CAT JUNCTION
JUMP
JUMP
POP
JUNCTION/
JUNC/JUNC
JUNC/OF
JUNC/END

Input Form of Semantic ATN

```
(MEASUREMENT/
   (GROUP
      (CAT SECTION T
           (SETR WHERE *)
           (TO MEAS/SECTION))
      (WRD (INPUT OUTPUT) T
           (SETR I/O LEX)
           (TO MEAS/I/O))
      (CAT MEAS/QUANT T
           (SETR QUANT *)
           (TO MEAS/QUANT))
      (CAT JUNCTION T
           (SETR TERM *)
           (TO MEAS/TERM))
      (CAT TERM/TYPE T
           (SETR TERM *)
           (TO MEAS/TERM))
      (CAT TRANSFORMER T
           (SETR WHERE *)
           (TO MEAS/SECTION))))

(MEAS/SECTION
   (GROUP
      (WRD (INPUT OUTPUT) T
           (SETR I/O *)
           (TO MEAS/SECT/I/O))
      (JUMP MEAS/SECT/I/O T)))

(MEAS/SECT/I/O
   (GROUP
      (CAT MEAS/QUANT T
           (SETR QUANT *)
           (TO MEAS/END))
      (JUMP MEAS/END (GETR I/O))))

(MEAS/I/O
   (GROUP
      (CAT MEAS/QUANT T
           (SETR QUANT *)
           (TO MEAS/I/O/QUANT))
      (JUMP MEAS/I/O/QUANT T)))

(MEAS/I/O/QUANT
   (GROUP
      (CAT PREP T
           (TO MEAS/I/O/QUANT))
      (CAT TRANSFORMER T
           (SETR WHERE *)
           (TO MEAS/END))
      (CAT SECTION T
           (SETR WHERE *)
           (TO MEAS/END))
      (JUMP MEAS/END T)))

(MEAS/TERM
   (CAT MEAS/QUANT T
        (SETR QUANT *)
        (TO MEAS/TERM/Q)))

(MEAS/TERM/Q
   (GROUP
      (CAT PREP T
           (TO MEAS/TERM/PREP))
      (JUMP MEAS/TERM/PREP T)))
```

```
(MEAS/TERM/PREP
   (GROUP
      (CAT PART T
           (SETR WHERE (BUILDQ (+ *)
          TERM))
           (TO MEAS/END))
      (JUMP MEAS/END T
           (SETR WHERE (BUILDQ (+ (PREF  ))
          TERM
          (PART/RANGE TERM))))))

(MEAS/QUANT
   (GROUP
      (WRD (ON OF) T
           (SETRQ CLASSES (PART TERMINAL JUNCTION NODE SECTION))
           (TO MEAS/PREP))
      (WRD AT T
           (SETRQ CLASSES (NODE TERMINAL))
           (TO MEAS/PREP))
      (WRD (BETWEEN FROM) T
           (TO MEAS/BETWEEN))
      (WRD ACROSS T
           (SETRQ CLASSES (PART JUNCTION))
           (TO MEAS/PREP))
      (WRD IN T
           (SETRQ CLASSES (PART TERMINAL JUNCTION SECTION))
           (SETRQ I/O INPUT)
           (TO MEAS/PREP))
      (WRD THROUGH T
           (SETRQ CLASSES (PART TERMINAL JUNCTION SECTION))
           (TO MEAS/PREP))
      (WRD (OUT FROM) T
           (SETRQ CLASSES (SECTION))
           (SETRQ I/O OUTPUT)
           (TO MEAS/PREP))
      (JUMP MEAS/PREP T))
   (POP (BUILDQ (REFERENCE ((QUANT)
       +))
  QUANT)
       T))

(MEAS/PREP
   (PUSH CIRCUIT/PLACE/ T
        (SENDRQ NOPRO T)
        (SETR WHERE *)
        (TO MEAS/END))
   (PUSH JUNCTION/ T
        (SENDRQ NOPRO T)
        (SETR WHERE *)
        (TO MEAS/END))
   (PUSH PART/ T
        (SENDRQ NOPRO T)
        (SETR WHERE *)
        (TO MEAS/END))
   (CAT SECTION T
        (SENDRQ NOPRO T)
        (SETR WHERE *)
        (TO MEAS/END))
   (PUSH PRONOUN/ (GETR CLASSES)
        (SENDR TYPES (GETR CLASSES))
        (SETR WHERE *)
        (TO MEAS/END)))

(MEAS/END
   (POP (BUILDQ (MEASURE + + +)
  QUANT WHERE I/O)
       T))

(MEAS/BETWEEN
   (PUSH PRONOUN/ T
        (SENDRQ TYPES (NODE TERMINAL))
        (SETR NODE1 *)
        (TO MEAS/BET/N1))
   (PUSH CIRCUIT/PLACE/ T
        (SETR NODE1 *)
        (TO MEAS/BET/N1))
   (PUSH NODE/BET T
        (SETR WHERE *)
        (TO MEAS/END))
   (WRD OUTPUT T
        (TO MEAS/BET/OUT)))

(MEAS/BET/N1
   (WRD (TO AND) T
        (TO MEAS/BET/AND)))

(MEAS/BET/AND
   (PUSH CIRCUIT/PLACE/ T
        (SETR NODE2 *)
        (TO MEAS/BET/END)))

(MEAS/BET/END
   (POP (BUILDQ (MEASURE + + +)
  QUANT NODE1 NODE2)
       T))

(CIRCUIT/PLACE/
   (JUMP TERMINAL/ T)
   (JUMP NODE/ T)
   (WRD THERE T
        (SETR POPVAL (BUILDQ (PREF (NODE TERMINAL))))
        (TO POP/VAL/)))

(NODE/
   (GROUP
      (WRD (NODE N) T
           (TO NODE/1))
      (JUMP NODE/1 T)))

(NODE/1
   (GROUP
      (WRD (BETWEEN JUNCTION) T
           (TO NODE/BET))
      (CAT NODE T
           (SETR NODE *)
           (TO NODE/END))
      (CAT INTEGER (AND (IGREATERP * -1)
   (ILESSP * 27))
           (SETR NODE (PACK (LIST (QUOTE N)
      *)))
           (TO NODE/END))))

(NODE/BET
   (GROUP
      (WRD OF T
           (TO NODE/BET))
      (CAT SECTION T
           (SETR PART1 *)
           (TO NODE/BET/P1)))
   (PUSH PART/ T
        (SETR PART1 *)
        (TO NODE/BET/P1)))

(NODE/BET/P1
   (WRD AND T
        (TO NODE/BET/AND)))

(NODE/BET/AND
   (PUSH PART/ T
        (SETR NODE (BUILDQ (NODE/BETWEEN + *)
      PART1))
        (TO NODE/END))
   (CAT SECTION T
        (SETR NODE (BUILDQ (NODE/BETWEEN + *)
      PART1))
        (TO NODE/END)))

(NODE/END
   (POP (GETR NODE)
       T))

(TERMINAL/
   (GROUP
      (CAT TERM/TYPE T
           (SETR TERM/TYPE *)
           (TO TERM/TYPE))
      (WRD ITS T
           (TO TERM/ITS))
      (WRD OUTPUT T
           (TO TERM/OP))))

(TERM/TYPE
   (GROUP
      (WRD TERMINAL T
           (TO TERM/TYPE/2))
      (JUMP TERM/TYPE/2 T)))

(TERM/PREP
   (PUSH PART/ T
        (SETR PART *)
        (TO TERM/TERM))
   (WRD ONE T
        (SETR PART (BUILDQ (PREF  )
      (PART/RANGE TERM/TYPE)))
        (TO TERM/TERM))
   (JUMP TERM/TERM T
        (SETR PART (BUILDQ (PREF  )
      (PART/RANGE TERM/TYPE)))))

(TERM/TERM
   (POP (BUILDQ (+ +)
  TERM/TYPE PART)
       T))

(TERM/ITS
   (CAT TERM/TYPE T
        (SETR TERM/TYPE *)
        (TO TERM/ITS/END)))

(TERM/ITS/END
   (POP (BUILDQ (+ (PREF  ))
  TERM/TYPE
  (PART/RANGE TERM/TYPE))
       T))
```

```
(PART/
   (GROUP
      (CAT PART T
           (SETR PART *)
           (TO PART/END))
      (WRD (Q R D C) T
           (SETR TYPE *)
           (TO PART/ABBEV))
      (WRD LOAD T
           (SETRQ PART LOAD)
           (TO PART/END))
      (CAT SECTION T
           (SETR SECTION *)
           (SETRQ CLASSES (CAPACITOR DIODE RESISTOR TRANSISTOR
         ZENER/DIODE TRANSFORMER))
           (TO PART/SECTION))
      (JUMP PRONOUN/ T
           (SETRQ TYPES (PART)))))

(PART/ABBEV
   (CAT INTEGER T
        (SETR PART (PACK (LIST (GETR TYPE)
          *)))
        (TO PART/END)))

(PART/SECTION
   (TST RIGHT/TYPE (MEMB LEX (GETR CLASSES))
        (SETR PART (BUILDQ (FINDPART +  )
      SECTION LEX))
        (TO PART/END)))

(PART/END
   (POP (GETR PART)
       T))

(PRONOUN/
   (GROUP
      (WRD IT T
           (TO PRO/END))
      (WRD THAT T
           (TO PRO/THAT))))

(PRO/THAT
   (TST TYPE/CHECK (MEMB LEX (GETR TYPES))
        (SETR TYPES (LIST LEX))
        (TO PRO/END)))

(PRO/END
   (POP (BUILDQ (PREF +)
  TYPES)
       T))

(JUNCTION/
   (GROUP
      (CAT JUNCTION T
           (SETR JUNCTION *)
           (TO JUNC/JUNC))
      (WRD ITS (NULLR NOPRO)
           (TO JUNCTION/))))

(JUNC/JUNC
   (GROUP
      (WRD (JUNCTION CIRCUIT OF) T
           (TO JUNC/JUNC))
      (JUMP JUNC/OF T)))
```

```
(JUNC/OF
   (GROUP
      (CAT TRANSISTOR T
           (SETR TRAN *)
           (TO JUNC/END))
      (JUMP JUNC/END (NULLR NOPRO)
           (SETR TRAN (LIST (QUOTE PREF)
       (QUOTE (TRANSISTOR)))))))

(JUNC/END
   (POP (BUILDQ (+ +)
  JUNCTION TRAN)
       T))
```